LinShim6 - Implementation of the Shim6 protocol

`http://inl.info.ucl.ac.be/LinShim6`

Documentation - Version 0.6.x

Sébastien Barré
Université Catholique de Louvain
Belgium

Feb. 2008

Contents

1 Introduction

This document presents the UCL implementation of Shim6, for the Linux Kernel. It will evolve concurrently with the implementation. The aim is to present the implementation from three sides :

- The user point of view (implications on user programs, Shim6 API)
- The programmer point of view : interfaces between *LinShim6*, the kernel and the shim6d daemon.
- *LinShim6* internals.

An other goal of this document is to present current ongoing work, as well as parts that could be done by external developpers interested in joining the project.

The work presented here is based on my master's thesis[Bar06]. The thesis (written in French) discusses in details the first version of the implementation and can be downloaded at `http://inl.info.ucl.ac.be/publications/shim6-masterthesis`.

Like the whole project, this documentation is a work in progress. Every comment, suggestion of improvement, or proposal to participate is welcome. Comments regarding the code or the documentation may be sent to the mailing list : *shim6-impl@lists.gforge.info.ucl.ac.be* (subscription at `http://lists.gforge.info.ucl.ac.be/mailman/listinfo/shim6-impl`.)

The version documented here is 0.6. This (still partially) implements [NB07] and almost fully [AvB07] (only the keepalive option is not supported currently and the exploration method can be made more efficient).

Note that a complementary document [Bar07] gives a global overview of the architecture. You can also download it from the INL website.

2 The user point of view

Shim6 is a new sublayer inside IPv6. It is intended to be absolutely invisible by upper layers. Nevertheless, it could be useful for applications to specify that they either want or don't want to use the shim. Alternatively, some applications may want to have special control over the shim.

- **current situation** : Currently, applications aren't able to control the shim. When the first IPv6 packet is sent to a new destination, a Shim6 negotiation is triggered. However, you can easily implement your own heuristic for triggering a Shim6 context negotiation by modifying the file *shim6_pkt_listener.c*. The easiest is to specify another packet number threshold, but you can also add a timestamp to the packet listener, so as to have a time threshold, or use port information in the packet to make a decision.
- **the future** : The intent for the future is to support a Shim6 API, like the one specified in [KBSS07].

2.1 Installation

Installing the patch : Patch the kernel the usual way, then type `make xconfig`. The Shim6 option is available under Networking/Networking options/Shim6 support. It is currently not possible to compile it as a module.

Shim6 won't work with only the recompiled kernel. You will also need to install the LinShim6-x.y.z tarball. The LinShim6 daemon can be installed like any other package :

```
tar -zxvf LinShim6-x.y.z.tar.gz
cd LinShim6-x.y.z
./configure [--disable-debug] [enable-debug-kref]
            [--enable-log-expl-time]
    [--disable-cgacheck]
    [CPPFLAGS="-isystem /usr/src/linux/include"]
make
make install (as root)
cgad (as root)
shim6d (as root)
```

The configuration options are set by default to enable most debug messages (you are encouraged to let debug messages and report problems). The `log-expl-time` option may be enabled if you want to do measurements of exploration times. If enabled, several informations about explorations will be stored inside */etc/shim6/expl.log*. Informations are the locators used before and after the exploration, the exploration time (defined as the time interval between leaving and coming back to the operational state), and the number of probes sent and received.

The `disable-cgacheck` option allows to build a LinShim6 that will accept unsecured locator sets. That option exists only for interoperability tests (experimental phase) and should not be set otherwise. The local use of CGA by LinShim6 is not disabled by that option, however.

The last option may prove useful if the configure script cannot manage to find your Linux kernel headers.

2.2 The shim6d interface

Since version 0.4.3, the user interface with shim6d has been changed : Previously, a signal handler for SIGUSR1 created files with a dump of state information. This has been removed and replaced with a telnet server. This gives much more flexibility, for example one can control the shim6d daemon from a distant machine by connecting to port 50000. The currently available commands are :

- *ls* : List all states. States are named by their context tag written in hexadecimal form.
- *cat <state>* : Provides much information about a specific state, named with its context tag written in hexadecimal form.
- *rm <state>* : Deletes a specific context state, both in the daemon and in the kernel. Note that if the shim6d process is killed, every context state is automatically deleted.
- *quit* | *exit* : Close the telnet connection.
- *get timelog* : Only available if the `log-expl-time` option was enabled when calling `configure` for the package. Dumps the content of */etc/shim6/expl.log*.
- *set timelog* : Only available if the `log-expl-time` option was enabled when calling `configure` for the package. Deletes the *expl.log* file. This is useful for automating exploration measurement.

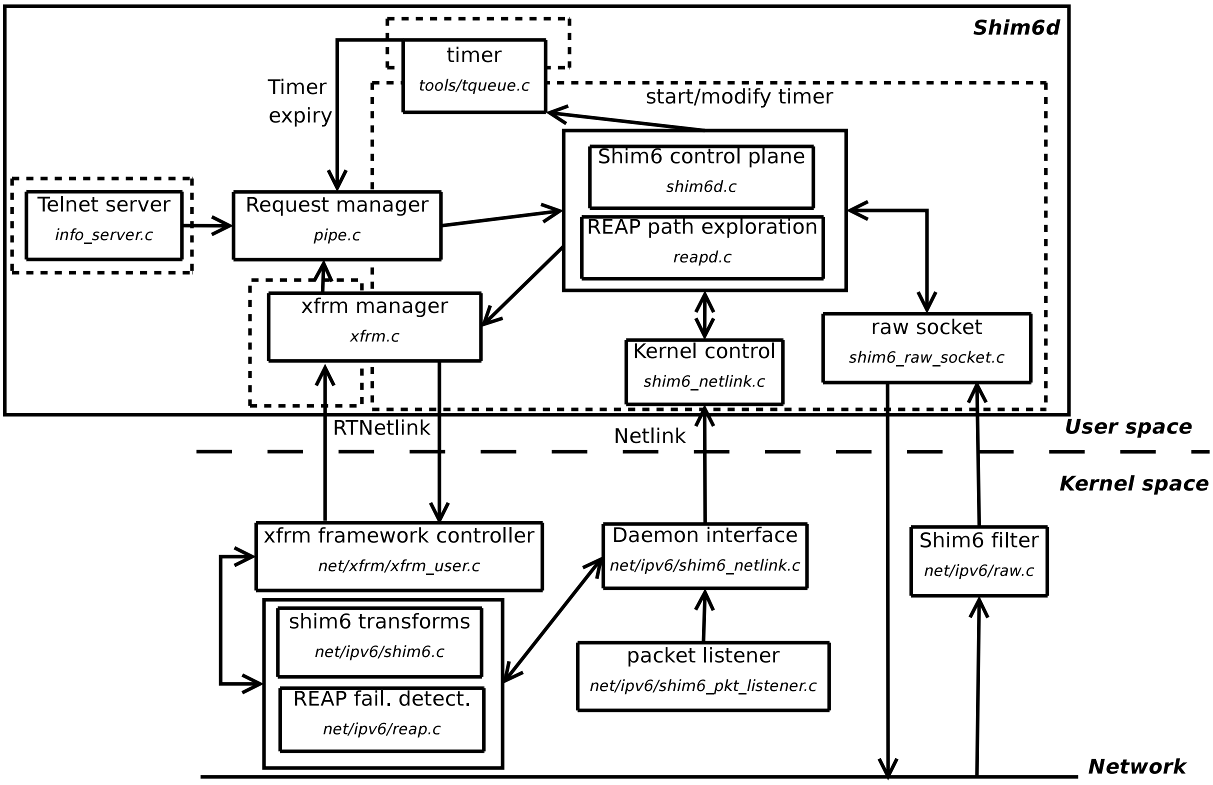

Figure 1: LinShim6 overall architecture

2.3 In case of problems

Error reports are very welcome and can be sent through the bug tracking system. You just need to follow the *Submit a bug report* link on the LinShim6 web page[1].

Error/info messages during execution are appended inside the */var/log/messages* file or another one, depending on your configuration.

3 LinShim6 overall architecture

In order to get a global view of the system, we invite you to first read [Bar07] before to proceed. For the sake of convenience, we reproduce here the big picture of the LinShim6 architecture (fig. 1).

Next sections will go deeper into the details of each part of that architecture. The upper part will be described in section 4. Section 5 introduces the *packet listener* module, responsible for deciding when and for which flow to start a Shim6 negotiation. Next the RTNetlink interface is described (section 6), so that we can follow with the details of the `xfrm` framework (section 7).

4 The shim6d daemon

Since version 0.5, almost all the *LinShim6* code has been moved to the daemon. This is why the previously called reapd daemon is now called shim6d (but it performs both Shim6 and Reap operations). Only failure detection and packet transformation are still done inside the kernel, for efficiency reasons (these two functions require work for each packet sent or received).

[1] `http://inl.info.ucl.ac.be/LinShim6`

4.1 Overview of the source files

The daemon provides several functions that are separated in different files :

- *main.c* : Main thread. It does the necessary work to become a daemon, initializes every module, then sits in an infinite loop, listening for three kinds of events, namely network messages (probes, keepalives, I1, R1, . . .), kernel netlink messages and pipe messages (requests from the other threads, see below).
- *shim6d.c* : performs every Shim6 related function (except address rewriting which is let to the kernel). The user space contexts are maintained there, inside a ULID hashtable and a ct hashtable (to look up either by ULIDs or by context tags). The main job of this module is to negotiate new contexts with peers, upon request by the kernel (four-way handshake).
- *reapd.c* : Implements the path exploration part of the REAP protocol. When a failure is detected by the kernel (send timer expiry) or an exploring probe is received, this module performs the exploration, and updates the `xfrm` context states when a new working locator pair is found.
- *raw_socket.c* : Tools for easily sending or receiving Shim6/Reap control packets. Only control packets are received, thanks to the filter present in the *linux_src/net/ipv6/raw.c* file of the kernel.
- *shim6_netlink.c* : Responsible for the Netlink communication with the kernel.
- *xfrm.c* : Communicates with the `xfrm` framework inside the kernel through the `RTNETLINK` API. The kernel side of the communication is implemented in *linux_src/net/xfrm/xfrm_user.c*. This module has a thread that listens to `xfrm` messages from the kernel. This thread is necessary by design of the `RTNETLINK` API.
- *info_server.c* : Runs in its own thread. This is the implementation of the *LinShim6* information server, waiting for telnet connexions on port 5000.
- *pipe.c* : Manages the transmission of requests from some threads to the main thread. This system has been designed to improve concurrency management. Before version 0.5, many semaphores were used to protect shared data, accessed concurrently from the timer or the *info_server* thread (now also the `xfrm` thread). But this was a useless complexity. We have then introduced the rule that any shim6 data (contexts and hash tables) may only be accessed by the main thread. The mechanism used is explained in section 4.2.

4.2 *pipe.c* : Using a pipe to serialize access to shared data

Since only the main thread may access the shim6 data, we need to be able to ask for some service from the other threads. For example, if some user types *ls* on the telnet console, the hash tables must be accessed to list the currently available contexts.

Instead of directly accessing the data, the *info_server* pushes a request on the pipe, by calling :

```
pipe_push_event(PIPE_EVENT_INFO_SRV,command);
```

Thanks to the `select()` system call, the main thread may be awoken by any event among network messages, netlink messages or pipe requests. Upon reception of a pipe request, `pipe_run_handler()` is called (*pipe.c*) and the message is dispatched to the correct handler. Then the command is executed inside the handler, that is, from the main thread. This provides a natural way of serializing execution, and allows for the removal of many mutexes.

Note that delegating actions to another thread implies sometimes the need to wait for the action to complete, before to do anything else. This is the case of the info server. If you type `ls`, the info server thread will ask the main thread to perform the listing action. But the prompt cannot be displayed before the listing action completes. Thus after pushing the request to the pipe (`pipe_push_event()`), the info server thread waits for a synchronization signal from the main thread (`pthread_cond_wait()`). When the listing action is done, the main thread sends the synchronization signal (`pthread_cond_signal()`), resulting in the info server thread displaying the prompt again.

5 Triggering a context establishment

Currently the context establishment trigger doesn't use the `xfrm` framework. It is implemented as a separate module, *shim6_pkt_listener.c*, that does something similar to connexion tracking : For each outgoing packet that is the first of an exchange, an entry is inserted inside a hashtable. It is removed if no packet is seen for the same exchange during more than one minute.

If the trigger condition is met, a `SHIM6_NL_NEW_CTX` message is sent to the daemon (app. A.3), so that a Shim6 negotiation is triggered. Currently the only supported trigger condition is the number of packets exchanged, it is configured by default to one packet.

By modifying the file *shim6_pkt_listener.c*, one can quite easily implement its own heuristic for triggering a Shim6 function. Currently the only heuristic is to trigger a negotiation after *nb_pkts_trigger* packets has been sent or received. This variable is currently set to one. Furthermore, this heuristic only triggers a context establishment if an outgoing packet is seen, in order to avoid a third party to be able to make the host create Shim6 states for arbitrary address pairs (which would be a potential DoS attack).

Once the decision has been taken to trigger a context negotiation, the kernel just sends a `SHIM6_NL_NEW_CTX` netlink message to the Shim6 Netlink multicast group.

6 Sending requests to the kernel : RTNetlink

In figure 1, we can see that the xfrm manager of the *LinShim6* daemon communicates with an `xfrm` framework controller through the RTNetlink interface.

In our implementation, we use a library written by Alexey Kuznetsov that communicates with the `xfrm` framework controller (*net/xfrm/xfrm_user.c*) through message passing over netlink[SKKK03]. The mapping from message number to message handler can be found at line 2033 of *xfrm_user.c* (in kernel 2.6.23).

When `xfrm` policies or states (explained later) must be created, their description is first created in user space inside the daemon (*src/xfrm.c*), then passed to the RTNetlink interface and interpreted by the *xfrm_user.c* file. The real creation of `xfrm` entities is thus implemented inside *xfrm_user.c*.

7 The XFRM framework

The kernel space part of *LinShim6* has completely changed since version 0.5 of *LinShim6*. The hashtables and shim6 contexts have disappeared from the kernel (they are now in user space only), and have been replaced in the kernel by the `xfrm` architecture.

Now only the context trigger, the address rewriting and the failure detection are done in kernel space, because each of these parts need actions to be taken for each packet coming in or going out.

In the next subsections we introduce the `xfrm` framework and our use of that framework for the specific purpose of Shim6 implementation. You can find additional documentation on that framework in [YMN+04], where one of the main authors the Linux IPv6 stack explains the general design ideas. The application of the framework for IPsec is described in [KME04], its application for Mobile IPv6 is explained in [MN04], and its application for Shim6 is described here, as well as in [Bar07] where a general overview is given.

7.1 Introduction to policies and states

Because IPsec makes uses of an SPD (Security Policy Database) and SAs (Security Associations), the `xfrm` framework also works with policies and states.

Packets going out first go through a *policy lookup*, in order to determine the output path that the packet must follow. For example if some flow needs AH and ESP transformations, the policy associated to that flow will specify that the output path must be set to `ah6_output()` → `esp6_output()` → `ip6_output()` (if the address family is IPv6). In the case of Shim6, the policy will specify that the output path must be `shim6_output()` → `ip6_output()`.

A policy is applied to a flow if its selector matches the flow. A selector has the following structure (*linux_src/include/linux/xfrm.h*) :

```
/* Selector, used as selector both on policy rules (SPD) and SAs. */

struct xfrm_selector
{
        xfrm_address_t  daddr;
        xfrm_address_t  saddr;
        __be16  dport;
        __be16  dport_mask;
        __be16  sport;
        __be16  sport_mask;
        __u16   family;
        __u8    prefixlen_d;
        __u8    prefixlen_s;
        __u8    proto;
        int     ifindex;
        uid_t   user;
};
```

Selectors allow to match packets against addresses, ports and protocols, as well as ranges thanks to the masks. If some field in the selector is set 0, it is interpreted as 'any'.

For the case of Shim6, selectors are defined as follows (*daemon_src/xfrm.c*).

Outgoing packets : All fields are 'any', except :

- the addresses : <ULID_local, ULID_peer>
- the user : `getuid()`
- the family :`AF_INET6`

Incoming packets : Also the addresses, user and family are the only filled in fields. Strangely enough at first sight, the <src,dst> address pair is set to <ULID_peer,any>. This is to be able to efficiently reuse the existing `xfrm` hashtables, with minimal extensions to support Shim6, as explained in section 7.3.

If there is a match between a given packet header and a policy, then the path of the packet is appropriately changed. We specify the path of a packet in the IPv6 stack by using a **template** vector. A template describes a given transformation. For example if we want to successively apply AH, ESP then Shim6 transformation[2], entry 0 of the template array would describe the AH transformation, entry 1 the ESP transformation and entry 2 would describe the Shim6 transformation.

The last part of the xfrm framework is the state, historically known as **security association**. A security association maintains all the state necessary for performing a given transformation. It is also unidirectional, because of the design of the IPsec protocol.

The Shim6 Security Associations : In the case of Shim6, the outbound transformer must be able to perform failure detection and ULIDs to locators rewriting. It thus need to contain the ULID pair, the Locator pair and the Context tag that will be inserted in the Shim6 header in case of transformation. A flag is also present to specify that address rewriting is needed or is not. Similar data is maintained in the reverse Security Association for performing the reverse translation. Note that the context tag stored in the outbound SA is the peer context tag (written to outbound packets), while the one stored in the inbound SA is the local context tag (expected in received packets from the peer). We also need to store the REAP failure detection timers. They are stored in a memory area whose pointer is shared by the inbound and outbound SA. We need to do that since the Keepalive and Send timers must be started when packets flow in one direction, and stopped when they flow in the other direction.

[2] This is an example, currently the implementation only allows Shim6 transformations. Extension to IPsec support should be simple, however.

The next sections will describe with more details the way policies, selectors, templates and Security Associations are dealt with in the case of *LinShim6*.

All `xfrm` operations by the daemon part of *LinShim6* are implemented in *src/xfrm.c* (in the daemon tarball).

7.2 `xfrm` policies

A policy embodies a *selector*, a `direction`, an `action` and a `template`. After the shim6 negotiation by the daemon terminates, the first thing the daemon does is create the kernel part of the shim6 state, which consists in two policies and two states. We explain here the role of policies.

A policy is represented by a structure defined in *include/net/xfrm.h* :

```
struct xfrm_policy
{
        ...
        struct xfrm_selector     selector;
        struct dst_entry        *bundles;
        u16                      family;
        ...
        u8                       action;
        ...
        struct xfrm_tmpl         xfrm_vec[XFRM_MAX_DEPTH];
}
```

The policy structure is filled in initially by `xfrm_add_policy()` (*net/xfrm/xfrm_user.c*), upon reception of a `XFRM_MSG_NEW_POLICY` message from user space.

In the above listing, we find the selector, the action, and the template vector. These three things make the core of the policy and will be explained hereafter in this section. The family is simply `AF_INET6` in our case. Note that the direction is not part of the structure, because policies are stored in different tables according to their direction.

The possible directions are `XFRM_POLICY_OUT`, for outgoing packets, or `XFRM_POLICY_IN`, for incoming packets. These two policies are actually quite different.

Outbound policy : The outbound policy serves to detect the packets when they are still in the transport layer, and change their outgoing path, in such a way that they go through the Shim6 layer. For Shim6 packets, our intention is not to filter out packets, thus xfrm *action* is always `XFRM_POLICY_ALLOW`. The selector is set to match with the **identifiers** (ULIDs), not the locators. This is because we are still above the Shim6 layer. A *template* must also be assigned to the policy (policy creation by the daemon is done in `xfrm_add_shim6_ctx` (*src/xfrm.c*)). The template provides a description of the "transformation" that will be performed. Note that in the context of `xfrm` with Shim6, the word "transformation" is used for address rewriting, but also timer updates for REAP failure detection, even if no actual transformation is performed. This is because we use the `xfrm` "transform" mechanism to perform both functions.

The Shim6 daemon defines the templates in `create_shim6_tmpl()` (*src/xfrm.c*). The template indicates that the address family is IPv6 and the protocol is Shim6. Note that we store 32 low order bits of the context tag in the spi field of the template. This is to get efficient context-tag based lookup of Shim6 context, and will be explained later.

The policy lookup for outgoing packets is done in the transport layer, right after the routing table lookup (see fig. 3). For the case of TCP, the function of interest is `inet6_csk_xmit()` (*net/ipv6/inet6_connection_sock.c*): First a check is done to see if a routing cache entry is present. Note that the routing cache entry is also an xfrm cache and thus XFRM policy lookup is only necessary when the first packet is sent (or when the policy is changed). The routing table is consulted by `ip6_dst_lookup()` and immediately after the xfrm policy table is consulted by `xfrm_lookup()`. Note that UDP does the same thing in `updv6_sendmsg` (*net/ipv6/udp.c*), and the raw sockets implement it in `rawv6_sendmsg` (*net/ipv6/raw.c*). This part of outgoing packet processing is represented in the upper-right part of figure 3.

Let's now dig into the `xfrm_lookup()` function (file *net/xfrm/xfrm_policy.c*). If no policy is cached for that particular socket, then a general policy lookup is performed (`xfrm_policy_lookup()`). The

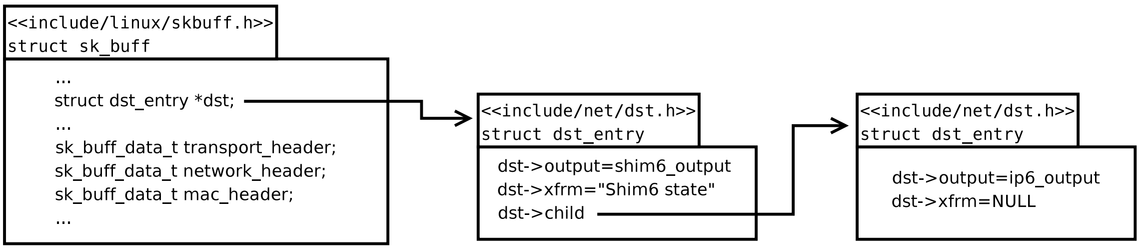

Figure 2: Linked list of output functions

lookup uses a hashtable and verifies a matching between the packet header and the selector we have configured previously. If no matching policy is found, xfrm processing is now finished. If a policy is found, then the corresponding template is resolved, leading to the construction of a *bundle of transformations*. Currently *LinShim6* only supports one transformation for a given flow (future work includes supporting for example chaining AH or ESP transformation with Shim6). In the case of Shim6, The bundle contains only one entry, for Shim6. As indicated previously, the outgoing path of the packet must be changed. The outgoing path is described by a linked list of `dst` structures. Each such structure contains a pointer to a `dst_output()` function. After a basic routing table lookup, the linked list only contains one entry, which points to `ip6_output()` (*net/ipv6/ip6_output.c*). Figure 2 shows the state of the linked list after the execution of `xfrm_lookup()`.

Figure 2 also shows that a dst structure contains an entry called `dst->xfrm`. This entry is of type `struct xfrm_state*`, and points to the real Shim6 state, that must have been created before, since the `xfrm_lookup` function only performs a lookup for such a state, it does not create it. The lookup is performed based on the same selector as defined previously. More details on `xfrm` states are given in section 7.3.

Inbound policy : The inbound policies are used differently from the outbound policies. The main difference can be summarized by saying that outgoing packets experience a *policy→state* sequence, while incoming packets experience a *state→policy* sequence. This can be observed by comparing the left and right part of fig. 3. For the outbound direction, the policy dictates the transformations that the packet will undergo. For the inbound direction, we receive a packet with a given order of extension headers. The extension headers are parsed, and a 'transformation vector' is built as the headers are parsed, so that the sequence of transformations is remembered. Next, `xfrm6_policy_check()` verifies that the observed sequence of headers were indeed authorized. The Shim6 inbound policy is only necessary to prevent Shim6 packets from being dropped by the `xfrm` subsystem.

In the case of Shim6, we currently only support one transformation (the Shim6 transform). Thus our inbound policy simply verifies that the received packet were a normal Shim6 packet, without any other transformation.

Shim6 policies are defined by the *LinShim6* daemon in `xfrm_add_shim6_ctx()` (*src/xfrm.c*). Note that the policies are defined with the ULIDs in the selectors, because policies are managed in the transport layer. They thus need not be updated (`xfrm_update_shim6_ctx()` upon update of the context (that is, change of the current locators). Nevertheless, `xfrm_update_shim6_ctx()` does make a policy update for the outbound direction only, because this triggers a routing cache flush as a side effect. This flush is necessary to force a new routing table lookup for the concerned sockets, since the new locators may need to go out through a different interface.

The `xfrm` implementation of policies is located in *net/xfrm/xfrm_policy.c*. In the next section, we will describe the state-related implementation, located in *net/xfrm/xfrm_state.c*.

7.3 `xfrm` states

Like the policies, `xfrm` states are created by the function `xfrm_add_shim6_ctx()` (*src/xfrm.c*). `xfrm` states are defined through a structure `struct xfrm_state` described in *include/net/xfrm.h*. Some parts of this structure are given below :

```
 struct xfrm_state
{
        /* Note: bydst is re-used during gc */
        struct hlist_node       bydst;
        struct hlist_node       bysrc;
        struct hlist_node       byspi;
        ...
        struct xfrm_id          id;
        struct xfrm_selector    sel;
        ...
        struct xfrm_lifetime_cfg lft;
        ...
        /* Shim6-related data */
        struct shim6_data       *shim6;
        ...
        struct xfrm_lifetime_cur curlft;
        ...
        /* Reference to data common to all the instances of this
         * transformer. */
        struct xfrm_type        *type;
        struct xfrm_mode        *mode;
        ...
        /* Private data of this transformer, format is opaque,
         * interpreted by xfrm_type methods. */
        void                    *data;
}

/* Ident of a specific xfrm_state. It is used on input to lookup
 * the state by (spi,daddr,ah/esp) or to store information about
 * spi, protocol and tunnel address on output.
 */
struct xfrm_id
{
        xfrm_address_t  daddr;
        __be32          spi;
        __u8            proto;
};
```

The hashtables : The three fields `bydst, bysrc` and `byspi`, are the collision lists of three different hashtables. Each hashtable permits retrieving an `xfrm` state with a different key (resp. dst, src or spi). The corresponding lookup functions, located in *net/xfrm/xfrm_state.c* are :

- `xfrm_state_lookup()` : Uses the `byspi` hashtable. The exact key used is the tuple `(daddr,spi,proto,family)`. Note that the contexts with null spi (`x->id.spi is 0`) are not inserted in that hashtable.
- `xfrm_state_lookup_byaddr()` : Uses the `bysrc` hashtable. The exact key used is the tuple (`daddr,saddr,family`).
- `xfrm_state_find()` : Uses the `bydst` hashtable. The exact key used is the tuple

With our LinShim6 patch, we define two additional lookup functions, necessary for proper operation of the Shim6 protocol.

- `xfrm_state_lookup_byct()` : Performs a context tag based lookup. Because we don't want to pollute kernel code with additional hashtables, we reuse the SPI hashtable (where of course, SPI does not mean Security Parameter Index). The SPI is 32 bits long, while the context tag is 47 bits long. Thus we store the 32 low order bits of the context tag in the SPI field of the Shim6 `xfrm` state. So that those bits are used as a key for finding the context. The `xfrm_state_lookup_byct()` function computes the hash and iterates over the collision list (that is, the `byspi` collision list) until the matching state is found. A state is considered to match if its family is `AF_INET6` protocol is `IPPROTO_SHIM6` the 47 bits of context tag match and the `xfrm` state is inbound.

- `xfrm_state_lookup_byulid_in()` : Performs a ULID-based lookup for inbound states. The outbound Shim6 states are found with the standard `xfrm_state_lookup_byaddr()` function. But for the inbound states, things are more complicated because the `xfrm` framework expects the source and destination addresses carried inside the incoming packet to be the identifiers for the context. Thus we must use the *locators* as `xfrm` identifiers if we want to use the standard functions. This is not acceptable since two different Shim6 states may use the same locator pair, and thus the locator pair is not a unique identifier.

  The solution is to store the identifiers in the `shim6` field of the structure `xfrm_state`, and create this custom lookup function. In order to still make an efficient lookup, we make use of the `bysrc` hashtable. The exact key used is the tuple `(saddr,daddr,family)`, where the source address is the remote identifier. However, we must set the destination address to `any (::)`, to avoid a conflict with the `byspi` hashtable, which uses that address as part of its key.

The two functions described above make possible for the same inbound `xfrm` state to receive packets, either with any locators and the Shim6 extension header (`xfrm_state_lookup_byct()`), or without the extension header and using the ULIDs as locators(`xfrm_state_lookup_byulid_in()`).

Selector and identifier : The next two fields are the selector and the identifier. The selector is the same as the one used for defining the associated policy. It is used to identify a state, but it is not sufficient, since the selector actually identifies a `range` of states (because of the possibility to define masks for addresses and ports). Thus the `daddr` field of the `xfrm id` is the particular address associated to that flow. Because of the one-one relationship between Shim6 policies and states, the `daddr` field from the selector and the xfrm id are the same.

Note that the `proto` field also exists in both the selector and xfrm id structures. Again, the meaning is different. In the selector, the proto field indicates that the upper layer flow must have the given protocol number. Shim6 sets this field to 0 (any), because `Shim6` associations are only based on network data, not at all on transport data. On the other hand the `proto` field of the `xfrm id` refers to the protocol number of the particular transformation performed with that state (Remember that each state is responsible for only one transformation, and thus a vector of states must be used if several transforms are to be applied). In the case of Shim6, we put there the Shim6 protocol number. We temporarily chose 61, since IANA has not yet given a number for the Shim6 protocol.

Shim6-related data : The `xfrm` states have a pointer to each possible transform data. Thus there are other fields for AH, ESP, ... (not shown here). Unused pointers are set to NULL. Our Shim6 structure is defined as follows (*include/linux/shim6.h*)

```
 /* shim6 data to be stored inside struct xfrm_state */
struct shim6_data {
        /* inbound - ct is ct_local
         * outbound - ct is ct_peer */
        __u64                ct;
        /* inbound - in6_local is ULID_local, in6_peer is ULID_peer
         * outbound - in6_local is lp_local, in6_peer is lp_peer */
        struct in6_addr      in6_peer;
        struct in6_addr      in6_local;

/* flags */
        __u8                 flags;
```

```
#define SHIM6_DATA_TRANSLATE 0x1 /* Translation activated */
#define SHIM6_DATA_INBOUND   0x2 /* context is inbound */
#define SHIM6_DATA_UPD       0x4 /* context update */
};
```

As shown in the comments, the content depends on whether the state is inbound or outbound. If the context is inbound, either `xfrm_state_lookup_byct()` will use the local context tag to find the state, or `xfrm_sate_lookup_byulid_in()` will use the identifiers. If the context is outbound, then we need to replace the identifiers (stored in the selector) with the locators, and insert the Shim6 extension header with the peer context tag.

The two first flags are self-explained. The third one is set to 1 by the daemon when communicating the structure to the kernel (`xfrm_update_shim6_ctx()` - *src/xfrm.c*), to tell the kernel that the given state is an update, not a new Shim6 context. This information is used in the kernel side by `shim6_init_state()` (*net/ipv6/shim6.c*).

lifetime : The `curlft` field keeps track of the number of bytes, packets, and seconds the state have seen. In Shim6 we are more interested in the lifetime in seconds, since we want to remove a state if it is no longer used. The time is updated in `shim6_input()` and `shim6_output()` (*net/ipv6/shim6.c*). The timeouts are configured by setting the `lft` field of that state. This is done through a message from user space. LinShim6 configures the timeout to `SHIM6_TEARDOWN_TIMEOUT` (defaults to 10 minutes) in `xfrm_lft()` (`src/xfrm.c`). As can be seen in that function, there exist a soft and a hard timeout. Both are managed in *xfrm_timer_handler()* (*net/xfrm/xfrm_state.c*). The hard timer automatically deletes the state and notifies the daemon, while the soft timer just notifies the daemon. In LinShim6 we only set the soft timer, because it is not sufficient to delete the state, we need to delete the inbound and outbound states and policies, as well as the daemon state. The expiry notification is sent through RTNetlink (see fig. 1), and handled by `xfrm_rcv()` (*src/xfrm.c*).

type and mode : Each `xfrm` transform is defined by a mode and a type. Currently 5 transform modes can be seen in `xfrm`. Each mode is defined by a `XFRM_MODE_*` constant in *include/linux/xfrm.h* and a specific file, *net/ipv6/xfrm6_mode_*.c*. The **mode** defines an input and an output function, that **modify the packet structure** appropriately. The **type** also defines an input and an output function, that **change the packet content** according to the particular transformation. Thus, the same mode may be used with several types if they need the same transformation.

For Shim6, we defined both a new mode and a new type. Our new Shim6 mode (*net/ipv6/xfrm6_mode_shim6.c*) performs a conditional packet structure modification. That is, if the `SHIM6_DATA_TRANSLATE` flag is set, then the locators differ from the ULIDs and space must be reserved for the Shim6 extension header. On the other hand, if the locators and ULIDs are the same, then no packet modification is needed.

The Shim6 type is defined in *shim6.c*. It registers the `shim6_input()` and `shim6_output()` functions, that perform the real work of the Shim6 sublayer. Those functions resp. call `reap_notify_in()` and `reap_notify_out()` (*reap.c*) for updating the REAP timers.

REAP timers : REAP timers are placed in the private `data` field of the state, because they need to be shared by the inbound and outbound state. We allocate a memory space for the REAP context when creating the outbound Shim6 `xfrm` state, then the inbound state is just linked to it (see `shim6_init_state()` (*net/ipv6/shim6.c*))

8 The path of a packet through the networking stack

The anchor points of *LinShim6* inside the Linux Kernel are as shown in figure 3. In this section we summarize the path of packets by describing this figure.

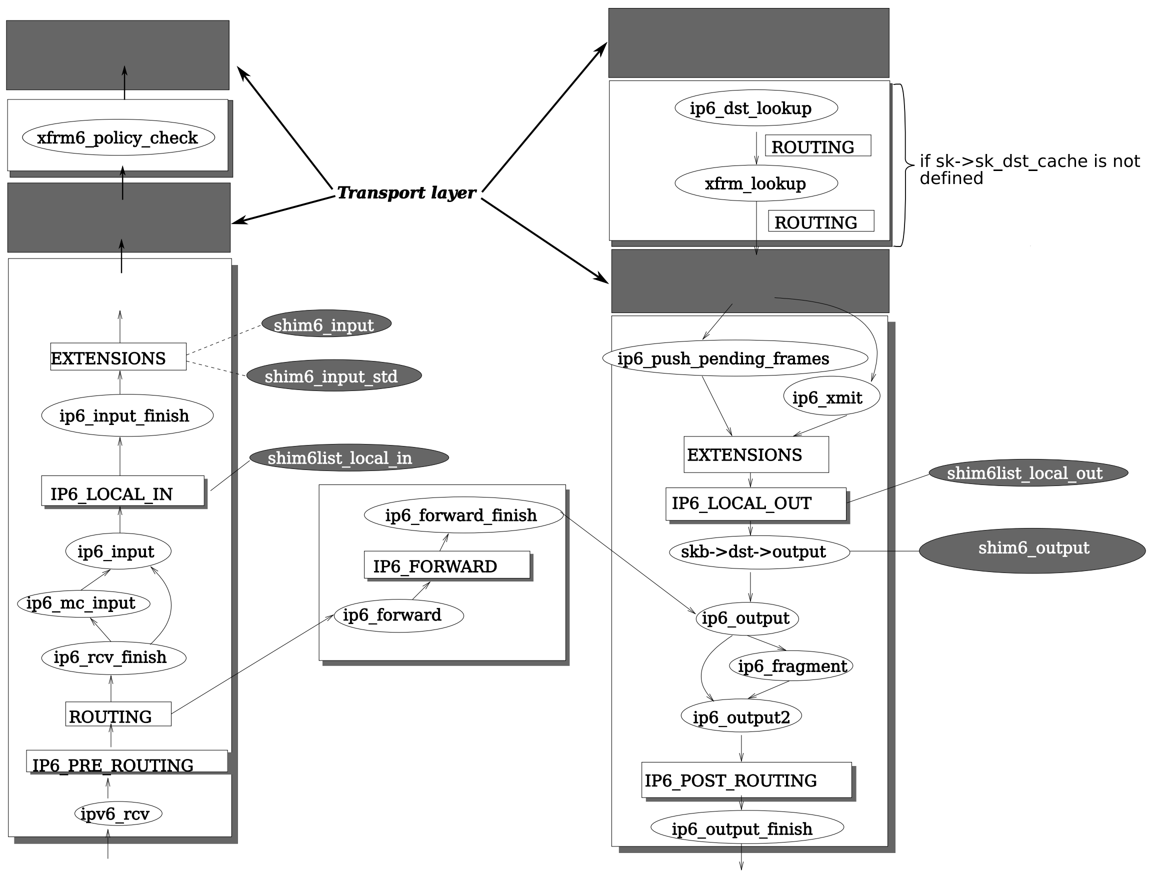

Figure 3: Packet path inside the IPv6 stack with *LinShim6*

8.1 Incoming packets

Incoming packets go through the Shim6 packet listener when hitting the `IP6_LOCAL_IN` netfilter hook. If an entry already exist for that packet, the corresponding counter is incremented, if not nothing is done (only outgoing packet may generate a new entry in that module).

Next the `ip6_input_finish()` function verifies if a raw socket is listening for that packet. It is the case of Shim6 control packets, that directly go through the Shim6 filter (and pass the filter), to finally arrive in the LinShim6 daemon, as indicated in the general architecture (fig. 1).

`ip6_input_finish()` then iterates over each extension header and calls the corresponding handler. The behaviour for Shim6 depends on whether the Shim6 extension header is present or not.

- **with the extension** : Like other extension headers, the Shim6 payload extension header is registered as an IPv6 protocol, so that it is dispatched the normal way by the 'resubmit' loop in `ip6_input_finish()`. In this case the `shim6_input()` function is called to handle the packet. This can be either a Shim6 control packet or a payload packet. In the first case, the packet is sent to the raw socket and interpreted by the daemon.

  In the later case, the payload extension header is used to match a context (`shim6_xfrm_input_ct()`), the addresses are translated and the packet is further processed by the 'resubmit' loop in `ip6_input_finish()`.

- **without the extension** : This case is more complicated since we need to match the packet against a potential Shim6 context, at the right step. When the extension header is present, we can parse the extension headers normally, and be sure that Shim6 will be managed at the right place. Without the extension, we need to do as if it were there.

  Thus, the chosen solution is to use the `shim6_input_std()` function for packets without extension headers, and insert some code in `ip6_input_finish()` to check when to send the packet to the Shim6 layer (relative to other extension headers), in case the Shim6 header is not present. The `shim6_input_std()` function in turn calls `shim6_xfrm_input_ulid()` in order to enter the xfrm framework, with a context lookup based on the ULIDs.

Later, in the transport layer, a call to `xfrm6_policy_check()` verifies that the transformed packet was indeed acceptable according to the local policies.

8.2 Outgoing packets

The first outgoing packet of a newly opened socket has no routing cache yet. For that packet only `ip6_dst_lookup()` finds the outgoing interface through the routing table. The source address is also chosen there according to RFC3484 rules[Dra03] if it was left unspecified by the application.

Right after that `xfrm_lookup()` checks if an `xfrm` policy exists for that flow, in which case it constructs the bundle of transformations corresponding to that flow. This determines the sequence of `skb->dst->output()` functions that will be called later. For Shim6, the output function is `shim6_output()`.

When hitting the netfilter `IP6_LOCAL_OUT` hook, the packet goes through the Shim6 packet listener module, that creates an entry for that flow. If the packet is the first one with this address pair (from any socket), the *LinShim6* daemon is notified to start a Shim6 negotiation (with the current heuristic). The other packets will just cause the corresponding counter to be incremented.

If a routing cache entry already exists for a given socket (see fig. 3, `sk->sk_dst_cache`), the calls to `ip6_dst_lookup()` and `xfrm_lookup()` are not needed. For that reason we must flush the socket caches when we change the current locators of a flow, since this may result in a change of the outgoing interface.

9 REAP Implementation

As previously mentioned, the REAP implementation is split in a kernel and a user space part. The division may be summarized by figure 4.

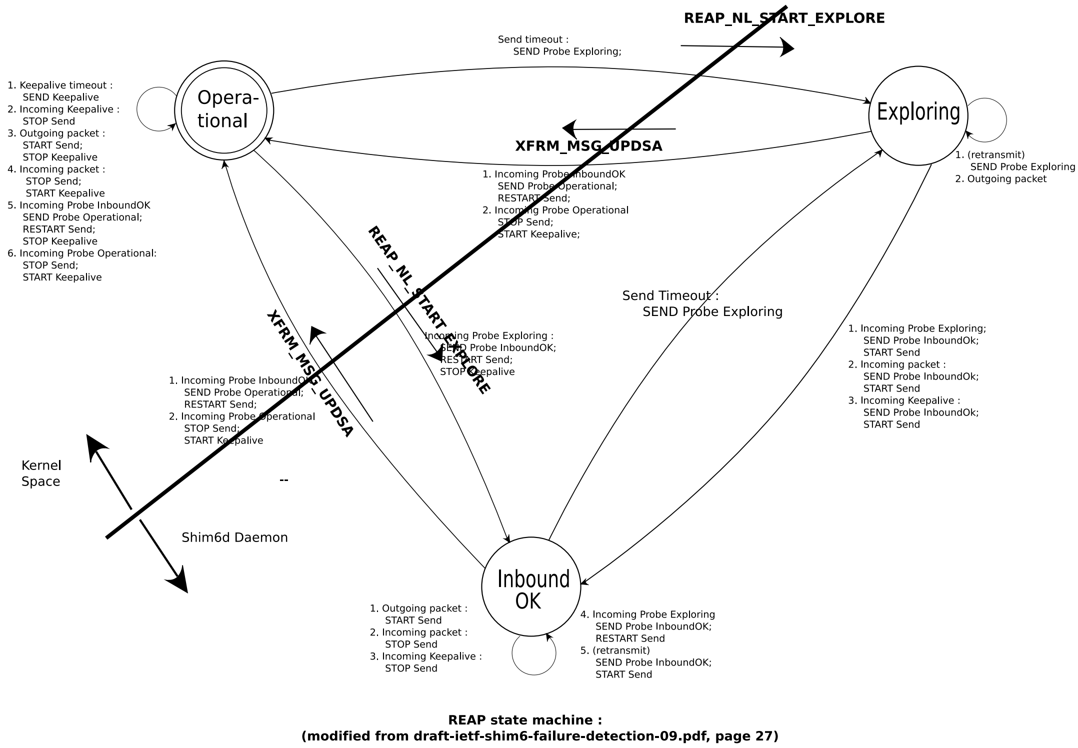

Figure 4: Interaction kernel/userspace for REAP

The REAP protocol runs in user space, almost independently from the kernel. The idea is to manage the send and keepalive timers in kernel space. Then, if the send timer occurs to timeout, the kernel informs the REAP daemon (by netlink), in order to start an exploration. When the exploration is terminated, the REAP daemon informs the kernel about the new operational address pair to be used.

The daemon also has its own send timer, which is used only during an exploration (while the kernel send timer is used only when the reap context is in state operational).

9.1 Triggering an exploration

Two things can trigger an exploration :

- Send timer expiry : This is detected by the kernel, which sends a netlink message `REAP_NL_START_EXPLORE` to the daemon. This results in the daemon starting the exploration process.
- Receiving a probe message : The probes are received by both the daemon and the kernel, so that it isn't necessary to send a netlink message from one to the other. The kernel just goes into inbound_ok state and adapts its timers, but lets the daemon perform the exploration.

9.2 Sending probes

The REAP context maintains a list of sent and received probe reports (with the locator pair used and the nonce of each probe). This list is kept during the whole time of an exploration. It is cleared either after the reception of probe operational or 10 seconds after the reception of a probe inbound ok. (because in the first case we are certain that the peer is operational, in the second we hope so, but this is not sure. That's why we keep the locators during 10 seconds after ending an exploration).

9.3 About (un)verified locators

HBA is not supported in versions 0.6.x. But CGA is supported since version 0.6.

We have used the DoCoMo SEND implementation as a starting point for CGA support in LinShim6. DoCoMo SEND provides a way to configure CGAs with different levels of granularity, ranging from one CGA PDS for the whole system, to a specific CGA PDS per address.

LinShim6 reuses the DoCoMo SEND configuration file for CGA parameters, thus benefiting from the same granularity. When starting, it registers every existing CGA address in the system, as well as every CGA PDS. However, only one PDS is sent to the peer for a given shim6 session, to be in accordance with the draft, Annex D.4. The chosen PDS is the one associated with the ULID pair used for that session. This means that LinShim6 is able to manage several PDS for different Shim6 sessions, while keeping exactly one PDS for a given session.

10 cgatool

Since Version 0.6.1, a new binary is available in the LinShim6 tarball and called cgatool. It is a tool originated from the DoCoMo SEND implementation, and integrated in the LinShim6 package. It has also been modified to better suit the needs of LinShim6. Note that the text of this section is almost completely taken from the DoCoMo SEND documentation, and is reproduced here for the sake of convenience.

10.1 CGA generation

When generating a CGA, use the `-g` or `--gen` command line argument. To generate, you must provide a key, an IPv6 prefix, and a CGA sec value. There are four ways to provide a key:

1. Provide a certificate with `-C` or `--certfile`.
2. Provide a PEM-encoded RSA key pair with `-k` or `--keyfile`.
3. Generate a RSA key on the fly with `-R` or `--rsa <bits>`. You must also provide a keyfile with -k to which to write the new key. Note that the number of bits for the key is a **mandatory** argument. If you fail to give it, you will receive an error *EVP_PKEY_assign_RSA() failed*.
4. Provide DER-encoded CGA parameters with `-D` or `--derfile`.

Provide an IPv6 prefix with `-p` or `--prefix <prefix>`.

Provide a CGA sec value with `-s` or `--sec <sec value>`.

When generating, you must also provide a derfile with `-D` to which to write the new DER-encoded CGA parameters.

Some examples:

- Provide the key from mykey.pem:
```
# cgatool -g -k mykey.pem -o myder -p 2000::  -s 1
```
- Provide the key from myder:
```
# cgatool -g -D myder -o myder -p 2000::  -s 1
```
- Generate from the example parameters provided in rfc3972:
```
# cgatool --gen -D rfc_example.params -o myder -p fe80:: -s 1
 fe80::3c4a:5bf6:ffb4:ca6c
```

The amount of time needed for CGA generation depends on the speed of your hardware and the sec value. You should choose the largest sec value your hardware and patience can reasonably handle. On a 2GHz Pentium 4, sec=1 usually takes just a few milliseconds, while sec=2 takes at least a few hours. The faster your hardware (and the more patient you are), the larger the sec value you can use. The largest possible sec value is 7. If you provide the key from a derfile, cgatool will use the modifier in the CGA parameters, and will not search for a new modifier. Once finished generating, cgatool will print the new CGA to stdout, and write the CGA parameters to the provided derfile.

10.2 Verification

You will ordinarily not need to manually verify CGAs. This functionality is provided for experimentation and sanity checks. When verifying, use the `-v` or `--ver` command line argument. To verify, you must provide the CGA to be verified, and the CGA's DER-encoded parameters. Provide the address with `-a` or `--address`, and the derfile with the `-D` or `--derfile` argument. For example:

```
# cgatool --ver -a 2000::2073:8e00:6d:aa09 -D myder
```

10.3 cgatool console

Run cgatool with the `-i` or `--interactive` command line argument. You can set all the arguments one-by-one, and use the *show* command to display current CGA context state. If you set `USE_THREADS=y` in *Makefile.config*, you can also use multiple threads to search for the CGA modifier in parallel. (Of course, this is only useful if you have a multi-processor and / or multi-core system[3]). Set the number of threads to use with 'thrcnt <num>'. While generating, cgatool will search a certain number of modifiers, and then check for interrupts (i.e. You can halt generation with ˆC). The number of modifiers searched between interrupt checks is called the batchsize. You can change this value with the 'batchsize <num>' command. The default batchsize is 500000.

Aknowledgements

This work has been funded by the FRIA (Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture, rue d'Egmont 5 - 1000 Bruxelles, Belgium).

I would like to thank John Ronan for the bug reports he provided, thus helping in the improvement of this work. Thanks also to Shinta Sugimoto and Miika Komu for the constructive discussion held on the usagi mailing list. The original idea to use the `xfrm` framework for Shim6 implementation has been suggested by Shinta Sugimoto. Masahide Nakamura also provided some explanations about the xfrm architecture. Junxiu Lu provided several bug reports and recently joined the project as a developer. Thanks to Marcelo Bagnulo and his research group for interesting discussions and comments about this work.

[3]This feature was present in DoCoMo SEND and the corresponding code has been kept in LinShim6. But it is not yet integrated nor tested

Appendix

A Shim6 control messages sent through Netlink

While `xfrm` has its own RTNetlink interface for communicating with user space, we still use our own Netlink channel for sending some messages from the kernel to the *LinShim6* daemon. Note that as integration with the `xfrm` framework continues, this interface may completely disappear in the future.

A.1 SHIM6_NL_NEW_LOC_ADDR : Announce the apparition of a new locator

```
 *  -----------------------------------
 * |      IPv6 addr. (128 bits)       |
 *  -----------------------------------
```

- **from kernel to daemon**
- **role :** Add a locator in the local locator list for the daemon. The body of the message is only the new locator.

A.2 SHIM6_NL_DEL_LOC_ADDR : Announce the removal of a local locator

```
 *  -----------------------------------
 * |      IPv6 addr. (128 bits)       |
 *  -----------------------------------
```

- **from kernel to daemon**
- **role :** Removes a locator from the local locator list in the daemon. The body of the message is only the locator.

A.3 SHIM6_NL_NEW_CTX : A new context must be created

```
*  ---------------------------------------------
 * |local ulid (128 bits) | peer ulid (128 bits) |
 *  ---------------------------------------------
```

- **from kernel to daemon**
- **role :** Announce to the daemon that the condition to trigger a Shim6 negotiation is met for the given ULIDs. Currently, this is sent by the packet listener module (*shim6_pkt_listener.c*).

A.4 REAP_NL_NOTIFY_IN : Incoming packet notification

```
 *  ---------------------------------------
 * |  local context tag (64 bits, 47 used) |
 *  ---------------------------------------
```

- **from kernel to daemon**
- **role :** Notifies the daemon that a packet belonging to the context with given context tag has been received. This is used only when there is an ongoing exploration process for the affected context.

A.5 REAP_NL_NOTIFY_OUT : Outgoing packet notification

```
*  ---------------------------------------
* |  local context tag (64 bits, 47 used) |
*  ---------------------------------------
```

- **from kernel to daemon**
- **role :** Notifies the daemon that a packet belonging to the context with given context tag has been sent. This is used only when there is an ongoing exploration process for the affected context.

A.6 REAP_NL_START_EXPLORE : Begin a new exploration

```
*  ---------------------------------------
* |  local context tag (64 bits, 47 used) |
*  ---------------------------------------
```

- **from kernel to daemon**
- The (kernel) send timer has expired. The daemon must start a new exploration. Note that a the daemon can also decide by itself to start an exploration, for example if a locator disappears (as is the case when the wire is unplugged) or an exploring probe is received.

A.7 REAP_NL_SEND_KA : Send a keepalive

```
*  ---------------------------------------
* |  local context tag (64 bits, 47 used) |
*  ---------------------------------------
```

- **from kernel to daemon**
- **role :** Asks the daemon to send a keepalive for the specified context. This is necessary because in operational state, the keepalive timer is maintained inside the kernel.

References

[AvB07] J. Arkko and I. van Beijnum. Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming. Internet Draft, IETF, July 2007. <draft-ietf-shim6-failure-detection-09.txt>, work in progress.

[Bar06] S. Barré. Développement d'extensions au Kernel Linux pour supporter le multihoming IPv6. Master's thesis, UCL, 2006.

[Bar07] S. Barré. Implementing SHIM6 using the Linux XFRM framework. In *Routing In Next Generation workshop*, Madrid, dec 2007.

[Dra03] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484, Internet Engineering Task Force, February 2003.

[KBSS07] M. Komu, M. Bagnulo, K. Slavov, and S. Sugimoto. Socket Application Program Interface (API) for Multihoming Shim. Internet Draft, IETF, July 2007. <draft-ietf-shim6-multihome-shim-api-03.txt>, work in progress.

[KME04] M. Kanda, K. Miyazawa, and H. Esaki. USAGI IPv6 IPsec development for Linux. In *International Symposium on Applications and the Internet*, pages 159–163, January 2004.

[MN04] K. Miyazawa and M. Nakamura. IPv6 IPsec and Mobile IPv6 implementation of Linux. In *Proceedings of the Linux Symposium*, volume 2, pages 371–380, July 2004.

[NB07] E. Nordmark and M. Bagnulo. Level 3 multihoming shim protocol. Internet draft, draft-ietf-shim6-proto-08.txt, work in progress, May 2007.

[SKKK03] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549 (Informational), July 2003.

[YMN$^+$04] H. Yoshifuji, K. Miyazawa, M. Nakamura, Y. Sekiya, H. Esaki, and J. Murai. Linux IPv6 Stack Implementation Based on Serialized Data State Processing. *IEICE TRANSACTIONS on Communications*, E87-B(3):429–436, March 2004.