Implementing SHIM6 using the Linux XFRM framework

Sébastien Barré, Olivier Bonaventure
Université catholique de Louvain (UCL), Belgium

I. Introduction

THE Shim6 proposal [1] by IETF to solve the multihoming problem for IPv6 is reaching a consensus, although some aspects are still criticized. It is important for the success of a solution to test it in real conditions, but this has not been done yet.

The Shim6 approach relies on an ID/locator split concept, where the mapping is done inside the end-hosts, thanks to a new shim sublayer located in the IPv6 part of the networking stack. If a host owns several locators, an application willing to connect to another end point will use one of them as Upper Layer IDentifier (ULID), and the Shim6 new sublayer will provide the ability to change the locators at will while keeping the identifiers constant (rewriting the source and destination address fields on-the-flight). Another protocol, REAP [2] (for REAchability Protocol), provides failure detection and recovery capabilities to Shim6. It is able to detect a failure and send probes to the available locator pairs until a new working path is found. Then the Shim6 layer is told to change the currently used locators and the communication can continue without any change in the application.

Because the approach allows one to change locators during an exchange, it is necessary to provide a mean for verifying that the used locators are actually owned by the peer. Shim6 can use two mechanisms for that : Hash Based Addresses [3] are a set of addresses linked together, so that one can verify that two addresses have been generated by the same host. Cryptographically Generated Addresses [4] are a hash of a public key and allow, together with a signature, to verify that the sender of a signed message is the actual owner of the CGA address used.

Since last year, we published evolving versions of the first publicly available Shim6 implementation, LinShim6. In this paper , we present the new architecture of the last version, 0.5[1]. While the previous one was a prototype implementation, this new version, a major rewrite, is designed to provide good performance and be easily extensible to support cooperation with other protocols that also use the XFRM architecture[5], [6]. Major improvements include careful separation between kernel and user space, cleaner insertion inside the kernel and better concurrency management.

In this paper, we start by presenting the XFRM framework. Then we describe our new design, highlighting the performance and modularity benefits obtained from the new architecture. Finally, we conclude the paper and discuss future directions.

Sébastien Barré is supported by a grant from FRIA (Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture), rue d'Egmont 5 - 1000 Bruxelles, Belgium

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

II. The XFRM framework

XFRM (for *transformer*) is a network programming framework included in the Linux kernel [7] to permit flexible transformation of packets. The framework obeys to a *Serialized Data State* model, as described in Yoshifuji et al. [8].

The idea is to be able to modify the path of packets through the networking stack based on some policies. The framework, originally designed to implement IPsec [5], has later been used for the Mobile IPv6 implementation [6].

A policy is made of a *selector*, a *direction*, an *action* and a *template*. The policy is applied to a packet if it matches the *selector* and is flowing in the *direction* of that policy (inbound or outbound). The selector mechanism allows one to use the addresses, ports, address family and protocol number as fields for the matching (see [9, sec. 4.4.1] for the precise semantics of a selector). Now let us assume that a packet matches a given policy. In that case the *template* is used to get a description of the tranformations needed for that kind of packet. Let us further assume that the packet needs AH (Authentication Header) and ESP (Encapsulating Security Payload) tranformation [9]. Then the corresponding states (one for AH and the other one for ESP) are found and a linked list of `dst` structures is created. A `dst` structure is normally the result of a routing table lookup, and contains information about the outgoing interface as well as a pointer to the function that must be called to send the packet (for example `ip_output` or `ip6_output`). As shown in figure 1, those structures may be linked together, so that several output functions are called sequentially.

After the `dst` path has been created, the linked list is cached for that socket, so that additionnal packets will flow through the IPsec layer as if it was part of the standard networking stack.

While outgoing packets are attached to one or several states by using `dst` entries (as shown in fig. 1), incoming packets are processed differently. Since those packets already have extension headers (for example AH and ESP), with lookup keys such as the SPI (Security Parameter Index), it is only necessary to lookup the XFRM states according to the information contained inside each extension header.

XFRM policies and states are created and managed from user space, with a *key manager* (as called inside the kernel) that communicates with the kernel part of XFRM by using the Netlink API [10].

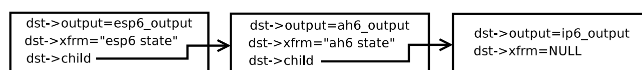

Figure 1. Dynamically created path for IPsec packets

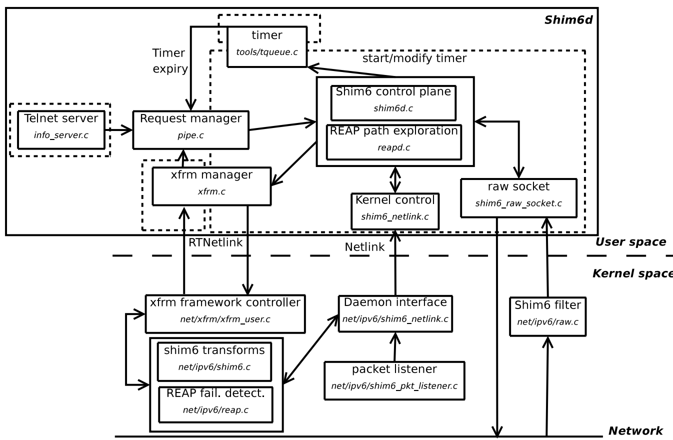

Figure 2. Shim6 overall architecture

III. LinShim6 0.5 : overall architecture

The Shim6 mechanism introduces a new sublayer inside the IPv6 layer, somehow like AH, ESP or Mobile IPv6. The flexible and modular, yet efficient XFRM framework thus fulfills particularly well the needs of the Shim6 transformations : after a Shim6 context (negotiated in user space) becomes established, XFRM policies and states are created, so that the packets matching the policies now go through `shim6_output()` (pointed to by `dst->output`, see fig. 1) and `shim6_input()` functions.

A global view of our design is given in figure 2. The upper part of the figure runs in user space as a daemon, and currently works with four threads (represented as dashed boxes). One is a `telnet server` that provides a Command Line Interface (CLI) to the daemon. The second one is the `timer` thread, that wakes up each time an expiration event happens. The `XFRM manager` listens to messages from the XFRM framework. Finally the main thread listens to messages from the network or the kernel, and reacts appropriately.

One of the major improvements of this new architecture is the insertion of a `request manager`. Its role is to simplify concurrency problems caused by external threads (that is, all threads except the main one) wanting to access Shim6 data structures (also used by the main thread). For example the `telnet` thread may want to dump the Shim6 states, or the `timer` thread may want to access the appropriate context before sending a probe. This led in previous versions to complex mutex schemes, that did not improve the concurrency, since each event was served by only a few lines of code.

A better scheme for avoiding concurrent access to critical data structures (contexts and hashtables) is to prohibit direct access to those structures from threads other than the main one. Instead, a generic `Request manager` has been written, so that external threads now send a request for service through a pipe. The main thread then performs the service as soon as it is ready. If several requests are sent concurrently, they are queued inside the pipe.

When a new Shim6 session starts, packets flow through the kernel and are counted by the `packet listener` module. This module uses the netfilter hooks `NF_IP6_LOCAL_IN` and `NF_IP6_LOCAL_OUT` to detect new exchanges and notify the daemon through the Netlink interface when a configured number (currently one) of packets has been seen for that flow. Note that, since Shim6 works at the IP layer, if several transport flows are started between two hosts, only one network flow is seen by the `packet listener` module. Later the `packet listener` could probably be integrated in the XFRM framework rather than use the netfilter hooks.

When the `shim6d` daemon is asked to create a new context, a four way handshake is performed, across the `raw socket`, attached to the protocol `IPPROTO_SHIM6`. An important point to note is that the same protocol number is used for control and data plane in the Shim6 protocol, which means that the raw socket would normally receive any data message equipped with a Shim6 extension header. For efficiency reasons, we prevent this by adding a Shim6 filter inside the raw socket implementation, as already done for several ICMPv6 messages (ICMPv6 filters may be configured from user space through the `ICMP6_FILTER` socket option).

Now suppose that the Shim6 (user space) context becomes established. We need to start the failure detection module, and thus make the packets go through the Shim6 transformers, `shim6_output()` and `shim6_input()`. Actually the transformer only performs address rewriting if ULIDs differ from locators. If not, it simply notifies the REAP `Failure detection` module that a packet has been seen. That module maintains the Keepalive and Send timers [2], and notifies the daemon if a failure has been detected. The result is that the `REAP path exploration` module starts sending probes across the `raw socket` until a new operational path has been found.

We decided to split the REAP protocol in two parts, respectively for kernel and user space. Again, this is for efficiency reasons : We try to keep as much as possible the protocols in user space, without sacrificing efficiency. But failure detection needs a timer to be updated for each packet sent or received, and thus *cannot* be implemented in user space.

Finally, when a new path has been found, the `XFRM manager` is notified to update the Shim6 XFRM states, so that the `Shim6 transform` module now adds the extension header and rewrites the addresses.

IV. Incoming packets

As other extension headers, the Shim6 extension header is registered in the kernel as a protocol. This allows the standard dispatching function of the Linux kernel (`ip6_input_finish()`) to direct the packet through the correct XFRM state.

The receiving process is illustrated in figure 3 : first the packet is sent to the raw sockets that are listening for that protocol number (left part of the figure). If the next header value in the IPv6 header is `IPPROTO_SHIM6` for example, the packet is delivered to the `raw socket` module of the daemon (if not filtered before).

Next, `ip6_input_finish()` enters a loop that parses each next header and calls the appropriate handler. If a Shim6 header is found, the corresponding handler is called and a context tag based lookup is performed to find an XFRM context.

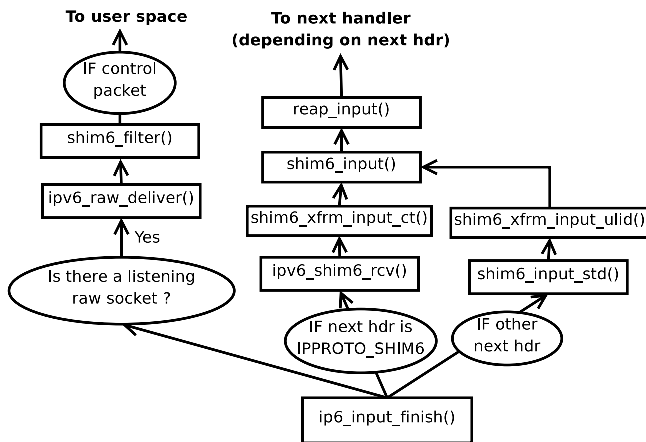

Figure 3. Incoming packet flow

Packets that do not contain the extension header also need to go through the `shim6_input()` function, since they may be using the ULIDs. In that case the standard dispatching function will not send the packet through XFRM, so we do that manually by calling `shim6_input_std()`. Actually in that case we do as if the Shim6 header was present, looking at what would be its place regarding extension header order, so as to go through Shim6 at the right step (see Nordmark and Bagnulo [1, sec. 4.6]). This case is shown in the right part of fig. 3.

Note that keepalives and probes are processed by both the `failure detection` (kernel) and `path exploration` (daemon) modules. In that case, the next handler (after function `reap_input()`) is `ipv6_nodata_rcv()` which simply terminates the processing since the next header is `IPPROTO_NONE` (59).

We extended the XFRM lookup functions to support ULID and context tag based lookups. The current XFRM framework maintains three hashtables : one uses the IPsec Security Parameter Index (SPI) as key [9], the second one uses the address pair and the last one uses a request ID (manually configured identifier used by IPsec). Rather than creating new data structures, we use the address pair based lookup for ULIDs, and SPI based lookup to find contexts on the basis of the context tag (the 32 low order bits of the context tag are used for that purpose). This way we can benefit from the performance of hashtable lookups, while keeping the existing data structures.

V. Keeping one context for each direction

Since IPsec works on a unidirectional basis, the XFRM framework only supports unidirectional contexts. For this reason, we split the previously unique Shim6 context into two XFRM contexts. The outbound context stores the peer's context tag and the locator pair (written in each outgoing packet), while the inbound context stores the local context tag and ULID pair.

But this solution is not sufficient for the failure detection module, which really needs a shared data area : when a packet goes out, a timer is started (Send timer). Thus we should maintain a timer structure inside the *outbound* context.

But the same timer is stopped when a packet comes in. It means that, when we have one context in hand, we actually need to get the corresponding reverse context also.

We solve this problem by using the private data pointer of an XFRM context. It is private in the sense that its meaning is not known by the XFRM framework and its usage is let to the particular instance of the transformer (Shim6). This allows us to do a reverse lookup at context creation only. After that, the shared memory area (only used by REAP) is accessible from both contexts. A reference counter is used to ensure that we free this memory only when the last XFRM context has been destroyed.

VI. Conclusion and future work

In this abstract, we discussed the current version of the LinShim6 implementation[2]. We explained design choices that should lead to best performance, while staying modular. Nevertheless, Shim6 processing is still an additional overhead and should be subject to measurements in real environments, especially to evaluate its impact on heavily loaded servers.

We emphasized the inherent flexibility of the XFRM framework. One of the benefits of using such a framework is that it permits the use of multiple mechanisms at the same time. For example, one could imagine using IPsec over Shim6 as suggested by Nordmark and Bagnulo [1]. This has still not been tested, however, and is kept for future work.

Our current work is to add HBA/CGA and attach live Shim6 machines to the IPv6 Internet. This will be reported during the workshop.

References

[1] E. Nordmark and M. Bagnulo, "Shim6: Level 3 Multihoming Shim Protocol for IPv6," Internet draft, draft-ietf-shim6-proto-09.txt, work in progress, October 2007.

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

[3] M. Bagnulo, "Hashed Based Adresses (HBA)," draft-ietf-shim6-hba-03.txt, work in progress., June 2007.

[4] T. Aura, "Cryptographically Generated Addresses (CGA)," RFC 3972 (Proposed Standard), Mar. 2005, Updated by RFCs 4581, 4982.

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

[6] K. Miyazawa and M. Nakamura, "IPv6 IPsec and Mobile IPv6 implementation of Linux," in *Proceedings of the Linux Symposium*, July 2004, vol. 2, pp. 371–380.

[7] K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler, *The Linux Networking Architecture*, Prentice Hall, 2005.

[8] 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*, vol. E87-B, no. 3, pp. 429–436, March 2004.

[9] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard), Dec. 2005.

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

[11] S. Barré, "LinShim6 - Implementation of the Shim6 protocol," Tech. Rep., Université catholique de Louvain, November 2007, http://inl.info.ucl.ac.be/LinShim6.

[2] A more detailed description may be found in [11]