FreeSWAN KLIPS version 2 requirements and implementation schedule

Michael C. Richardson
Sandelman Software Works Inc.
On behalf of FreeS/WAN
mcr@sandelman.ottawa.on.ca

September 16, 2001

1 Executive Summary

$Id: klips2req.tex,v 1.4 2001/09/17 03:37:26 mcr Exp $

This is a plan for evolution of current FreeSWAN KLIPS code into "next generation" KLIPS code. A review of requirements and applicability to intended design is presented, followed by a review of current KLIPS1 architecture.

The target architecture is then described.

A plan both to develope this document and to transition from old to new is presented.

2 Current KLIPS input/output structure

2.1 output: ipsec_tunnel_start_xmit

1. gather private information
2. clone skb if necessary
3. verify that packet is IPv4
4. compute hard header length
5. decrement TTL
6. lookup in erouting table
7. UDP port 500 exception
8. start encapsulation loop
   (a) check for DROP or missing eroute
   (b) check for REJECT eroute
   (c) check for PASS eroute
   (d) check for HOLD eroute
   (e) check for TRAP eroute, signal PF_KEY, swap to HOLD eroute
   (f) acquire lock for walking tdb chain
   (g) calculate headroom required for chain
      i. check if SA is in larval, drop
      ii. check if SA is dead, drop
      iii. check if replay overflowed, expire SA
      iv. check if lifetime counters have overflowed, expire SA
      v. switch on protocol type, to calculate headroom size.
         A. if ESP switch on protocol type to calculate tailroom size.
   (h) calculate mtudiff, send ICMP fragment needed. Mark "note2"
   (i) hack MSS if desired
   (j) copy upper (layer 2) header to safety if it was present
   (k) check if data fits in existing skb, else expand.
   (l) apply grouped transforms
      i. apply disaster of #ifdefs.
      ii. switch by protocol type, calculate headroom for this stage
         A. if ESP, then switch by cipher get headroom
         B. if ESP, then switch by hash to get tailroom
      iii. double check (not in NDEBUG) if there is enough headroom

iv. push the data ahead
v. double check (not in NDEBUG) if there is enough tailroom
vi. extend the data behind
vii. see if packet has become too long (bigger than 64K)
viii. finally move the plaintext as appropriate
ix. switch on protocol type
x. case: ESP
A. switch on cipher type, prepare IV
B. prepare self-describing padding
C. switch on cipher type, do encryption
D. switch on cipher type, update IV
E. switch on hash type, do authentication
xi. case: AH
A. prep replay info, headroom
B. switch on hash type, do authentication
xii. case: IPIP, apply encap
xiii. case: IPCOMP
A. call skb_compress
B. do some debugging
xiv. recalculate header checksum

(m) lookup eroute by new outer header, if we found something and the src/dst have changed

9. send ICMP if packet has become too big
10. re-apply link layer header if there was one.
11. attempt to re-route the packet
12. drop packet if new route leads to us again.
13. do connection tracking
14. do netfilter localout output call
15. call ip_send or IP_SEND depending on kernel version

2.1.1 Comments upon problems/limitations of transmit

2.2 input: ipsec_rcv

1. increment module use count
2. verify skb and data is not NULL
3. verify hard header length

4. clone (COW) if necessary
5. a number of poorly documented "assertions"
6. verify protocol number against packet and against protocol structure
7. verify that protocol is AH, COMP or ESP.
8. lookup each ipsecX device to determine which one has been bound to the receiving device. Grab ipsecprv device info.
9. if no device found, warn, but do not die
10. begin decap loop
    (a) lock tdb if this is first time through
    (b) verify that length is appropriate multiple if ESP
    (c) switch on protocol type, grab SPI value from appropriate place
    (d) format sa with satoa. (not found in code)
    (e) if AH, then determine AH header length, find next protocol value, and verify against expected length of AH header.
    (f) get spin lock if required
    (g) if IPCOMP
        i. check if IPCOMP is out most header, (not yet supported)
        ii. advance the tdb pointer and, if doing inbound policy check, then check SPI value. Complain if not matched.
        iii. decompress packet, reset ip header pointer to new value, loop (via continue)
    (h) lookup tdb based upon SA. `gettdb`
    (i) complain if no tdb
    (j) if doing inbound policy check
        i. check that outer source matches one on packet.
        ii. check that this tdb is the expected next from previous. (forward check)
        iii. check that this tdb expects to be attached to previous. (reverse check)
    (k) check if tdb state is larval, skip
    (l) check if tdb state is dead, complain
    (m) check lifetime (bytes - soft/hard, addtime - soft/hard, usetime - soft/hard, packet count - soft/hard). Expire TDB, tell pfkey if limit exceeded.
    (n) pick authlen, switch on auth type (MD5, SHA1)
    (o) switch on protocol type (ESP, AH only) and set up authenticator

(p) check sequence number to see if replay window rolled, if so expire

(q) check out replay window, dropping if it is a replay

(r) verify authenticator, check if there was authentication, switch on type

   i. MD5, call MD5Update and friends, checking if ESP or AH was involved

   ii. SHA1, call SHA1Update and friends, checking if ESP or AH was involved

   iii. none, do nothing

(s) check authenticator for NULL (which would imply not AH or ESP above)

(t) compare authenticator against hash, complain if failed

(u) update the replay window

(v) switch on protocol type

   i. if ESP

      A. switch on encryption algorithm

         - if 3DES, then find IV and set header length
         - otherwise, fail

      B. locate ciphertext based upon header length

      C. switch on encryption algorithm

         - if 3DES, verify data length multiple of 8 and decrypt.
         - no otherwise clause

      D. find next header type

      E. find padding

      F. verify padding

   ii. if AH, do nothing

(w) update protocol number in header (why?)

(x) switch on protocol type

   i. if ESP, the memmove as appropriate for ESP, skb_pull() to compact, and then skb_trim.

   ii. if AH, then memmove as appropriate for AH, skb_pull().

(y) update skb pointers to parts of packet.

(z) nuke any options that skb knew about, or skb-¿proto_priv (2.2+)

(a) recalculate the header checksum

(b) set the sbk protocol type to IP over ethernet

(c) advance tdb pointers

(d) if doing inbound policy check

   i. verify that backward policy agrees with forward policy

ii. check if next protocol field is not one we know about
A. complain that policy was not complete

(e) update ipcomp ratio counters if IPCOMP was involved, but this stage is not IPCOMP

(f) update the lifetime values in bytes, packets, and last used time.

(g) loop again if ESP, AH or IPCOMP

11. if original chain was IPCOMP, then advance tdb chain once (Why?)

12. if there is one last tdb

(a) verify that last protocol type was IPIP (no transport supported here)

(b) if doing inbound policy checks

i. advance tdbnext with inext, and complain if non-NULL. (i.e. check that this was last tdb)

ii. verify source IP address matches tdb source

(c) update lifetimes for this tdb

(d) if skb data len is too small for header length, complain

(e) pull up new header into skb

(f) advance ip pointer to inner header

(g) update raw header pointer

(h) zero protocol options

(i) update layer 2 protocol info to IP over Ethernet

(j) reset checksum info

13. if we are doing EROUTE checking (i.e. tunnel exit checking)

(a) setup for look up by src/dst in eroute table, checking for IPIP header.

(b) lock eroute table, lookup eroute

(c) record info we need and unlock

(d) if we found what we need, then lock, and lookup policy information by new said block.

(e) if no tdb found, then we drop packet

(f) walk policy_tdb chain, look for last one

(g) compare against tdb that we just used, complain if not the same.

14. unlock tbd

15. update stats if appropriate

16. release packet destination

17. if there was a layer 2, copy it back into place

18. do inbound policy checks if it was IPCOMP
19. do connection tracking
20. drop packet back into bottom half queue

2.2.1 Comments upon problems/limitations of receive

3 KLIPS1 static structure

3.1 klips1 radij

The `radij` module is an adaptation of the BSD `radix.c` code. It has been consistently renamed to as to coexist with `radix.c`. It implements a netmask aware patricia tree on blocks of data.

An explanation explains that the pronounciation is almost the same since the "j" is to be pronounced like the Greek chi.

3.2 klips1 eroute

An eroute entry describes an entry in the security policy database. The `eroute` currently only includes selectors for source and destination address.

3.3 klips1 tdbh

This is an array of pointers to `struct tdb` structures. It serves as a root for chained hash buckets. It is an open hash.

These are managed by the code in `ipsec_xform.c`. These provide a mapping from a `struct sa_id` to a `struct tdb` based upon SPI, protocol and destination address.

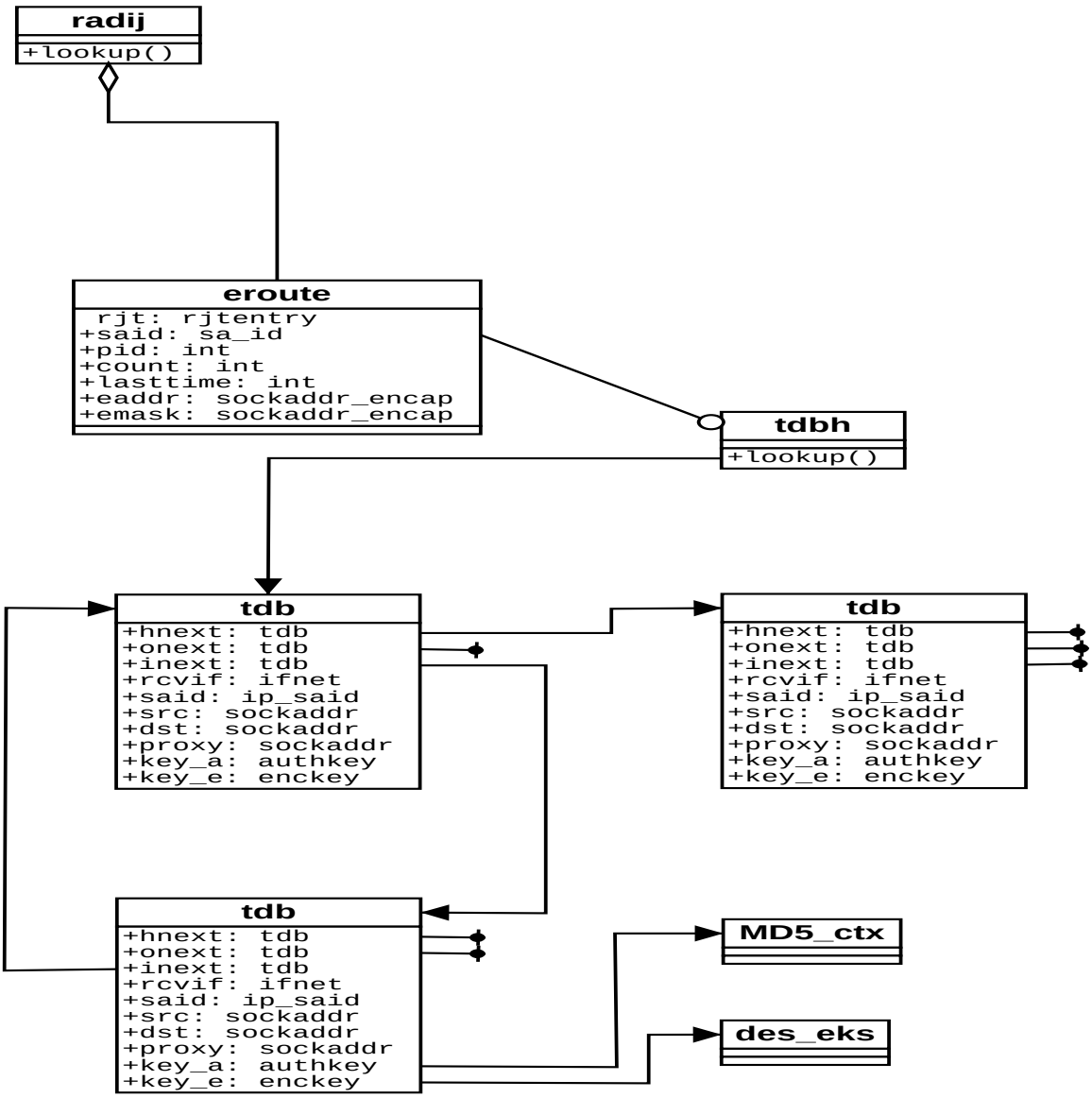
The entries in each bucket are linked together using the `tdb_hnext` field.

3.4 klips1 tdb

This structure represents all the information associated with a single transform. In the case that multiple transforms may be chained together, they are chained together using `tdb_inext` and `tdb_onext`. The "i" and "o" are short for inner and outer. Thinking of the resulting packet as an onion, these pointers describe transforms towards the **i**nner and **o**uter directions.

3.5 klips1 md5_ctx

This stores the key context for the MD5 authentication routines. This structure is different from the `MD5_CTX`, in that this is the HMAC version and contains an inner and outer `MD5_CTX`.


radij
+lookup()
eroute
rjt: rjtentry
+said: sa_id
+pid: int
+count: int
+lasttime: int
+eaddr: sockaddr_encap
+emask: sockaddr_encap
tdbh
+lookup()
tdb
+hnext: tdb
+onext: tdb
+inext: tdb
+rcvif: ifnet
+said: ip_said
+src: sockaddr
+dst: sockaddr
+proxy: sockaddr
+key_a: authkey
+key_e: enckey
tdb
+hnext: tdb
+onext: tdb
+inext: tdb
+rcvif: ifnet
+said: ip_said
+src: sockaddr
+dst: sockaddr
+proxy: sockaddr
+key_a: authkey
+key_e: enckey
tdb
+hnext: tdb
+onext: tdb
+inext: tdb
+rcvif: ifnet
+said: ip_said
+src: sockaddr
+dst: sockaddr
+proxy: sockaddr
+key_a: authkey
+key_e: enckey
MD5_ctx
des_eks

3.6 klips1 sha1_ctx

This block is not shown in the diagram, but serves an analogous function to `md5_ctx`.

3.7 klips1 des_eks

This block contains a DES key schedule (this is not equivalent to a DES key, but has been scheduled already). Note that this is really a container and is assumed to be of the proper size. The DES routines actually take a `des_cblock` as input.

4 Detailed Requirements

4.1 001: changeable gw wild-side addresses on-the-fly

4.1.1 001: Definition of requirement

Some systems use DHCP or IPCP (PPP) to get an address assigned. DHCP in particular has a clear lease time, and at the end of the lease, a different IP address may be assigned.

The requirement is for **FreeS/WAN** to interact with address assignment facilities, adjust lifetime parameters appropriately, and to transparently change systems with the address change.

4.1.2 001: response

The movement away from a layered device model means that KLIPS should not interfere with this process. Most of the issues should be on the key management side (**Pluto**).

This requirement is useful for Opportunistic Encryption.

4.2 002: address inertia

4.2.1 002: Definition of requirement

The essence of this requirement is that gateway's can remember where the wild-side address of road warriors are. Should a reboot (or a restart of **Pluto**) occur, it would re-initiate to these clients.

There are three levels of support which may be desireable:

Level 1 record only the wild-side address for re-initiation.

Level 2 record the wild-side address, and all current phase 1 (DH and SKEYID) keying materials.

Level 3 record the wild-side address, and phase 1 and phase 2 keying materials.

4.2.2 002: response

Satisfaction of level 1 of this requirement will require changes only to **Pluto**, specifically to provide a way to get a list of current connections, to record this in a stable file, and a for the boot up scripts to read the alternate list of configurations as well. So, this requirement can be satisfied without impact to KLIPS2 design.

Level 2 of this requirement has some issues. The storage of keying material on disk may be a source of concern. This issue would need to be addressed in the design. The source of this requirement is to provide reliable recovery and fast reboots, systems that involve operator intervention may not satisfy this. The chief advantage of storing the phase 1 information is that it reduces the

amount of time required to do DH exponentiation after a reboot. A new phase 2 would have to be done as well.

Level 3 of this requirement has further issues. It requires some help from KLIPS2 to provide for the retrieval of keying materials (including replay state) from the kernel, and subsequent reloading of it. There are clearly even more issues with making sure that the materials are not inappropriately revealed. In addition, the state of eroutes, filtering, etc. will need to be captured. Saving of this information may have very strong advantages in the opportunistic case, as the information on whether or not to set up an opportunistic tunnels is valuable as well. Further, in the opportunistic case the risk of disclosure of the keying material may be considered low enough that storing it is worthwhile.

In all three cases, there is a cost-benefit analysis to do, weighing the improvements in reliability and performance against the risks of inappropriate disclosure. The answer to this analysis may always be a local matter.

In addition, all three cases would apply to restarting of **Pluto** either on purpose (to facilitate easy updates), or due to program error (core dump).

There are further legal issues. Access to the keying materials may facilitate cooperation with law enforcement access. This is not regarded as a feature.

Opportunistic encryption would benefit from any amount of key maintenance. Road warriors are the ones most likely to benefit as they are turned off/suspended most often. However, their wildside address is also most likely to change, rendering any saved state that they have useless.

4.3 003: mini-database of road warriors that persists across reboots

4.3.1 003: Definition of requirement

This requirement appears to be a repeat of requirement 001.

4.4 004: connection up, down, wanted

4.4.1 004: Definition of requirement

All internal SA entries should have a status of whether the connection is up (keying material is available), down (keying material has expired), or is wanted (keying material not yet available). This is part one.

There is an additional situation in which the MAST device may need to be marked down. This is when it is known by the routing system that all routes to the outer destination will fail. This will typically only be true for systems without default routes (i.e. that are in a default-free zone). This second feature is part two.

4.4.2 004: Response

Part one is committed to. There is an issue that we currently do not look for expired SAs unless we are attempting to use them. To fix this, we will need to

walk the SA table periodically.

Part two raises some design questions. Specifically, how does one know if the outer destination is routable unless looks?

- each SA (and thus each conn) could maintain a pointer to a struct dst_entry. This has some savings in that one doesn't have to lookup the route each time that the SA is used. (One does the lookup if the entry is either invalid, or non-existant, this is just a cache. The TCP PCB does this as well)

  As these structures are reference counted, we can safely hang on to this.

  If asked about link status of a MAST device, then one just has to walk all SAs associated with this device, looking for at least one with SA which has not been obsoleted.

- alternatively, we do as we currently do, but upon failure to find a route to the outer dst, we bash the link status of the device to down. (We only change when all SAs say down, which makes this somewhat difficult)

  Once the device is down, then we should really discard any packets that arrive at the MAST device. We do not want to waste time encrypting things we would then through away.

  We could do something like let 1% through to do the above test, but that seems like a poor choice, since routing daemons may have found other ways around in the meantime, so no traffic would ever reach us.

The first solution is preferred, but neither are committed to at this time.

4.5 005: routing below tunnel layer to support mobility and multi-homing

4.5.1 005: Definition of requirement

Ciphertext packets should be treated like all other packets, and should be subject to normal routing proceedures.

4.5.2 005: Response

Implementation of MAST devices and packet escalator should accomplish this.

This is a critical piece for reliability of opportunistic encryption, as current routing tricks cause plaintext packets to blackhole after a physical interface flap.

4.6 006: SAs entries should be capable of overlapping

4.6.1 006: Definition of requirement

Currently klips1 apparently identifies a tunnel by what **remote** subnet it serves. That means that if a new tunnel is brought up serving the same subnet, it supersedes the previous one.

A more complex semantic is required, and a way to express it:

- sometimes you do want the new tunnel to supersede the old one.
- sometimes you want the new tunnel to operate in parallel, using equal-cost multipath, for load sharing.
- sometimes you want the new tunnel to just sit there in standby mode, to be used later
  - for fail-over
  - for mobililty

4.6.2 006: response

This is a misfeature, and is hereby deprecated.

Rollover of SAs is necessary for functional long-term opportunism.

4.7 007: why do equalizing schedulers not play well with tunnels?

4.7.1 007: Definition of requirement

linux/net/sched/sch_teql.c says:

> 1. `Slave devices MUST be active devices, i.e., they must raise the tbusy signal and generate EOI events. If you want to equalize virtual devices like tunnels, use a normal eql device.`

A normal "eql" device (`linux/drivers/net/eql.c`) simply does a simple form of round-robin scheduling among devices. It can round robin among devices of equal weight.

The teql scheduling device uses the scheduling system's back pressure (the `dev->tbusy` flag) to give each device as much as it wishes.

4.7.2 007: response

The KLIPS1 `ipsecX` devices are never busy. They therefore can not be equalized using this scheduler.

The virtual devices could use the `tbusy` mechanism. To be able to do this, the `MAST` device will have to be given a clear amount of resources on a per-virtual device basis. As the limit to throughput will be the lesser of encryption throughput and physical device throughput, once the buffers are full, the virtual device can raise the `tbusy` flag.

For this to be useful, the paths to the remote host must be different. Specifically, the outer destination address must in some way be different. If there are simply two physical ways to get to the same destination address then standard load-balancing would work once the `MAST` devices have processed the cleartext.

The use of the `tbusy` feature is not considered to contribute strongly towards Opportunistic Encryption. The creation of the MAST device is however critical.

4.8 008: decouple SA retrieval from DADDR (don't care how it arrived)

4.8.1 008: Definition of requirement

The desire is that the SA lookup by SPI should not depend upon destination address. This is in contradiction to IPsec requirements, but an implementation that assigns unique SPI numbers to all protocols can satisfy both requirements.

This requirement also helps NAT traversal in some cases.

4.8.2 008: response

As SPI values are assigned by the receiver for unicast, we can in general accomplish this by allocating all SPI values from a single pool. This is committed to.

This is neutral towards Opportunistic Encryption.

4.9 009: SPIs unique, independant of protocol and DADDR

4.9.1 009: Definition of requirement

This is a variation of requirement #8.

4.10 010: routing above tunnel layer

4.10.1 010: Definition of requirement

see JSD documents.

4.11 011: granularity smaller than host

4.11.1 011: Definition of requirement

It is desired to do SPD lookups based upon UDP/TCP port numbers (source and destination) as well as source and destination address.

In addition, for hosts a lookup based upon Unix UID should be possible.

For gateways, it is also desirable to do lookup based upon IPSO labels.

4.11.2 011: response

UDP/TCP port number lookups will be present in the second release of KLIPS2.

IPSO labels will be present in the third release of KLIPS2.

This feature will not contribute to Opportunistic Encryption.

4.12 012: /dev/ipsecNNN devices that could be chown(1)ed and chmod(1)ed.

4.12.1 012: Definition of requirement

One of the grand ideas of Unix is the notion that "everything is a file".

As a result, network devices don't show up in /dev/ in a useful way, and they don't have file-modes and file-owners. Instead you need to deal with them using special commands like **ifconfig**, and special system calls like **bind, setsockopt, ...**

As a result, it is clear how to establish an IPsec connection from host A to host B, but it is really not obvious how to establish an IPsec connection from user UX (process PX) to user UY (process PY).

Could a user have his own ipsec.conf file? How would that file be related to the system's ipsec.conf file?

Even if the user doesn't have his own ipsec.conf file, how do we implement per-user or per-process tunnels? I can imagine what the kernel code looks like to enforce the restrictions, but what does it look like to the user process? Making it look like a named pipe with a file-owner and some file-permissions is one way... that makes it look more like good-old "core" unix but less like other networking stuff.

4.12.2 012: response

Constructive proposals would be most welcome.

This feature is not committed to in any form at this time.

4.13 013: process to process tunnels

4.13.1 013: Definition of requirement

The idea here goes something like this: User IDs are a clumsy form of identification and authentication. Modern systems do much better.

For instance, using ssh, one can have one window (one process group) for which one has started an ssh-agent which holds one's certificates.

One can have another window logged in under the same user ID, but with different certificates, or indeed no certificates at all.

This is the modern approach: security resides in the keys, not in the user IDs.

User IDs have a clear meaning locally. They have only a weak relationship on relationship to entities on a distant system.

Certificates do have meaning as anyone with a public key in a trusted store with clear authorizations attached can trust some entity that has the corresponding private key.

4.13.2 013: response

During the definition stage of "granularity finer than per-host", (such as in requirement #11) it will be necessary to clearly articulate what the appropriate size of a "grain" will be.

No specific programming work is committed to for satisfaction of this requirement, rather a step in the process for definition of requirement #11 has been made.

4.14 014: ways to manipulate tunnel perms.

netfilter, pf_key, ioctl, */dev/ipsecNNN*

4.14.1 014: Definition of requirement

This is a variation of the question #13: how do user applications request security services from the kernel.

4.15 015: KLIPS as a loadable module (isn't it already?)

4.15.1 015: Definition of requirement

The KLIPS system should permit compilation as a loadable module to permit easy intregration.

4.15.2 015: response

This requirement is currently met in KLIPS1, and will be retained for future work.

While LKM is a permitted configuration, it must always be possible to compile the module statically.

4.16 016: stats: (number,time_of_last) packets (out,good_in,error_in)

4.16.1 016: Definition of requirement

Each SA bundle should maintain a list of statistics. This should include:

1. number of packets encapsulated
2. time last used to encapsulate
3. number of packets decapsulated properly
4. time last used to decapsulate properly
5. number of packets decapsulated improperly
6. time last used to decapsulate improperly

4.16.2 016: response

These numbers will be maintained. Appropriate locking granularity needs to be determined.

This feature must be designed in properly for it to work, and is therefore of priority.

4.17 017: integrate IPsec and firewall policy into Security Policy.

4.17.1 017: Definition of requirement

To provide industrial-strength security, the IPsec Security Policy Database should be integrated with the regular Linux firewalling facilities, specifically into the Netfilter/IPtables code.

Integration provides a single place to express policy. It prevents duplication of code: this is both a savings in physical quantities (kernel time and code space) as well as a reduction in opportunities for errors.

4.17.2 017: response

This is a core design requirement for KLIPS3.

4.18 018: full inbound policy checking

4.18.1 018: Definition of requirement

Upon decryption/decapsulation of a packet, the inner set of selectors should be checked against SA definition. This is a requirement from RFC2401. It provides a paranoid check against possible mis-behaviour/mis-configuration of a corresponding peer.

4.18.2 018: response

This will be present in the 1.92 release.

This facility will be present in the KLIPS2 design as an optional feature.

4.19 019: secure ciphers and hashes

4.19.1 019: Definition of requirement

The project will use only ciphers and hashes which are known to be secure. The definition of secure used includes: that the algorithm has sustained a reasonable length of analysis, that it can not be brute forced with current technology in less than a decade, and that there are no covert channels.

4.19.2 019: response

Use of MD5, SHA1 as hashes and 3DES for cipher is currently used. This will be maintained.

1DES is considered insecure and will not be added. RC4 is considered insecure and will not be added.

The project will only add new algorithms after extensive discussion. AES is currently under discussion, but is not scheduled to be added at this time.

4.20 020: kernel implementation (should be faster)

4.20.1 020: Definition of requirement

Appropriate benchmarks are sought to live up to.

4.21 021: plays well with routing daemons

4.21.1 021: Definition of requirement

In order to provide for appropriate interaction with routing daemons, a logical interface is required to which to anchor routes. The keying status of the connection should be reflected in to the link status of the logical interface.

4.21.2 021: response

The proposed MAST device satisfied this requirement, and is a defining feature of KLIPS3.

4.22 022: free of export restrictions

4.22.1 022: Definition of requirement

Many countries have placed restrictions on what their citizens may do, or having done this, may restrict who the product may be given to. The project's code should not be encumbered in this manner.

4.22.2 022: response

The project will continue to refuse contributions from residents or citizens of the United States of America. The recent changes to the administration of the BXA are not sufficient cause change this policy as they may change again very quickly.

4.23 023: standard crypto api to add newer ciphers and hashes

4.23.1 023: Definition of requirement

The current KLIPS1 encapsulation and decapsulation routines make explicit synchronous calls to the 3DES encrypt and decrypt functions. This causes three problems:

1. it makes it difficult to add new algorithms, both at compile time and at runtime.
2. it fails to make use of multiprocessor systems effectively
3. it fails to interface nicely to hardware acceleration devices

A standard API from FreeSWAN KLIPS to algorithm functions (e.g. 3DES-MD5-ESP) would provide for plug and play capabilities for algorithms.

An asynchronous interface would permit multiple processors or hardware accelerators to interface easily as well.

Despite this, the packets must still emerge from the system in the same order that they arrived. That is, they must not be reordered, as this causes inefficiencies for TCP.

4.23.2 023: response

A design to use an asynchronous interface to algorithms will be provided as part of KLIPS2.

The design proposed by Bart Trojanowski (rsa1) ¡bart@jukie.net¿ will be used as a basis.

4.24 024: opportunistic

4.24.1 024: Definition of requirement

4.25 025: SADB hash table will be locked for additions/deletions

4.25.1 025: Definition of requirement

4.26 026: use a refcount on each SA to increase locking granularity

4.26.1 026: Definition of requirement

5 Proposed data architecture

5.1 klips2 mast

The MAST device provide a mooring point for routing protocols and firewall policies. The MAST device represents one or more IPsec tunnels.

Packets routed to a MAST device will get encrypted with a default SA. A different SA may be specified using NetFilter (aka ipchains, aka iptables) rules.

Packets that are received via one of the SAs associated with this device will be marked as having been received on a MAST device.

The link status of a MAST device reflects the keying status of all underlying SAs. If at least one SA is keyed, then the MAST device will be up. If no SA that is associated has valid keys, or if no SAs that are associated with the device, then the device will be marked down.

This is provided in part to satisfy requirement 21 (see 4.21) and requirement 5 (see 4.5).

5.2 klips2 radij

The radij tree provides a lookup on eroute entities.

While the eroute database provides the IPsec security policy database (SPD), the radij tree is the index into it by selectors.

It is a BSD radix.c derived table. It has been modified to permit masking of bits of the index to occur in arbitrary places, thus permitting both source and destination addresses to be masked.

The radij tree will be replaced or augmented to support additional selectors. These include UDP/TCP port numbers, SPI numbers (for multiple layers of gateways), ICMP types and codes, and possibly also IPSO labels.

As opportunistic encryption creates large numbers of fully specified source/destination pairs, it will be investigated if an auxiliary table could provide a more efficient storage of these tables. Specifically, it is possible that connection tracking can more efficiently provide this required index.

5.3 klips2 eroute

The eroute is the object that represents an entry in the IPsec SPD.

This structure will be extended with additional selectors to support requirement 11 (see 4.11).

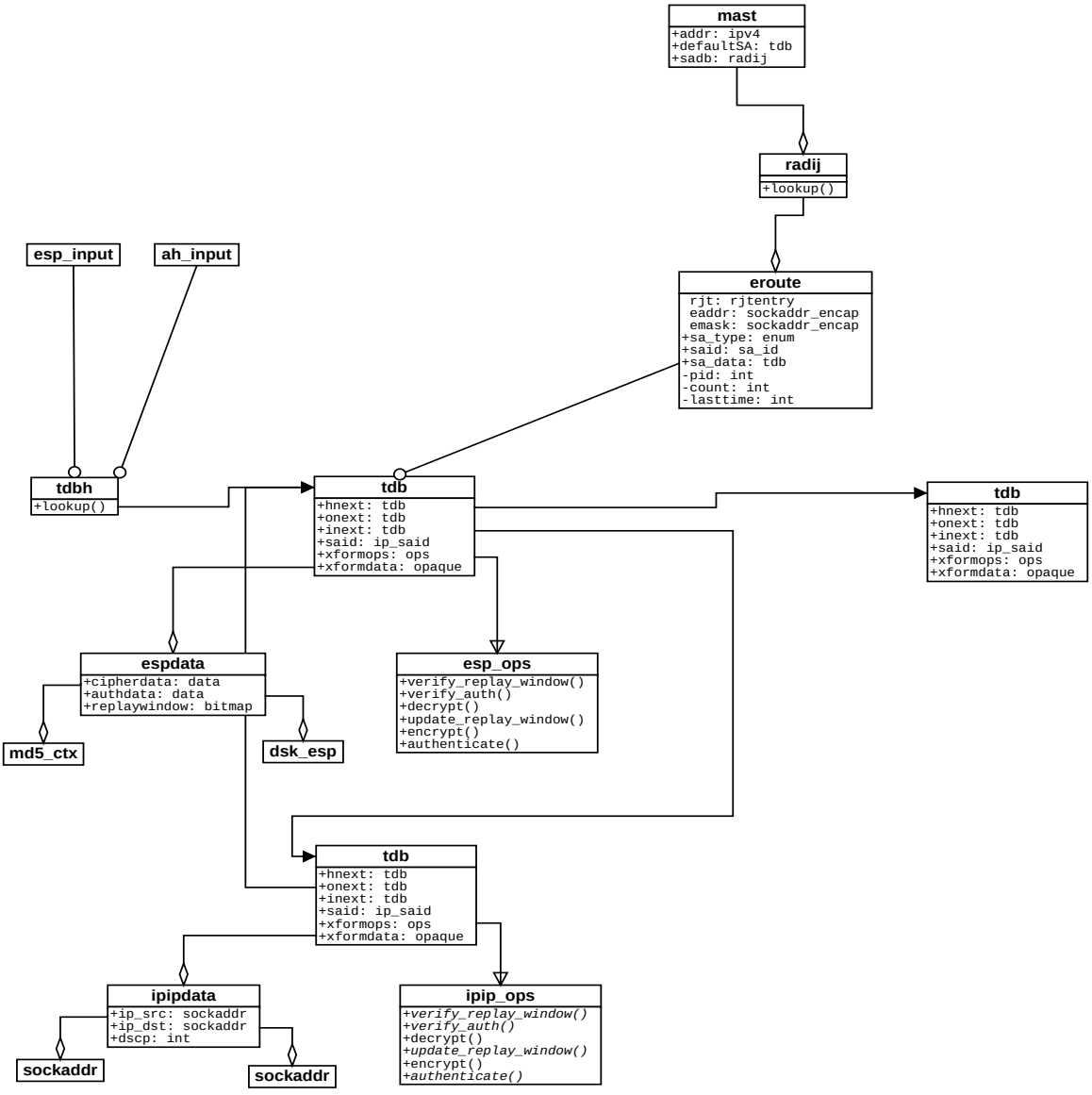
5.4 klips2 ipsec_sa_buckethead

This is the root of the transform database hash table.

This structure is the renamed `tdbh`.

5.5 klips2 esp_input

The current `ipsec_rcv` will be cloned to create an ESP receive function.


mast
+addr: ipv4
+defaultSA: tdb
+sadb: radij
radij
+lookup()
esp_input
ah_input
eroute
rjt: rjtentry
eaddr: sockaddr_encap
emask: sockaddr_encap
+sa_type: enum
+said: sa_id
+sa_data: tdb
-pid: int
-count: int
-lasttime: int
tdbh
+lookup()
tdb
+hnext: tdb
+onext: tdb
+inext: tdb
+said: ip_said
+xformops: ops
+xformdata: opaque
tdb
+hnext: tdb
+onext: tdb
+inext: tdb
+said: ip_said
+xformops: ops
+xformdata: opaque
espdata
+cipherdata: data
+authdata: data
+replaywindow: bitmap
esp_ops
+verify_replay_window()
+verify_auth()
+decrypt()
+update_replay_window()
+encrypt()
+authenticate()
md5_ctx
dsk_esp
tdb
+hnext: tdb
+onext: tdb
+inext: tdb
+said: ip_said
+xformops: ops
+xformdata: opaque
ipipdata
+ip_src: sockaddr
+ip_dst: sockaddr
+dscp: int
ipip_ops
+verify_replay_window()
+verify_auth()
+decrypt()
+update_replay_window()
+encrypt()
+authenticate()
sockaddr
sockaddr

5.6 klips2 ah_input

The current `ipsec_rcv` will be cloned to create an AH receive function.

5.7 klips2 ipsec_sa

This is the renamed `tdb` structure.

5.8 klips2 espdata

This is a per-SA structure that records all the information needed by the algorithm database functions to perform their functions.

5.9 klips2 esp_ops

This is the set of pointers to functions needed to perform the ESP steps.

5.10 klips2 md5_ctx

This is a subpiece of the espdata structure that stores MD5 keys.

5.11 klips2 ipipdata

This structure provides information needed by the IPIP transform.

5.12 klips2 ipip_ops

This is the set of pointers to functions needed to perform IPIP steps.

6 Milestone definitions

6.1 settle on project plan/milestones

The definition of the project plan (this document) and milestones (this section) must be agreed to. That is, the schedule for developing the schedule is itself subject to planning. This is this section.

6.2 settle on requirements

The list of requirements for low level rework must be agreed to.

The initial goal is to get the requirement section (**??**) completed by September 9, 2001.

6.3 target architecture

The target architecture is a set of diagrams and explanatory text. It should be agreed to by September 15, 2001.

6.4 transition plan

The transition plan explains which work will done first and how it will interact with other components.

The technical transition plan will be completed by August 31, 2001.

6.5 publish algorithm database design

A design for use of the algorithm database will be done as part of the target architecture milestone.

6.6 algorithm database work

6.7 ipsec_rcv rework

Work on redoing the receive functions using the algorithm database will begin in September and will be completed and tested for a release in November. This will have no externally visible changes to customers, the major goal of this stage is readability.

This work will consist of the following steps:

1. rename "tdb" structure to "ipsec_sa", add reference count.
2. create lifetime structure, moving all lifetime checks to common code.
3. create an ident structure to be friendlier to PF_KEY.
4. create transform data structure and ops structure containing a pointer to function for each each existing case statement of all "switch" clauses.

5. create per-packet state structure ("job") using the `skb->options` area and appropriate macros.

6. split up into pre-crypto, crypto and post-crypto stages, calling each one directly. (i.e. not through queues or callbacks yet)

7. use of low-level generic algorithm functions (e.g. 3DES, SHA1, MD5) to perform crypto.

8. a later effort will transform the receive function into the software path of the target architecture.

6.8 creation of MAST device

The MAST device will be introduced in September as an experimental device. Select code from the transmit side will be placed into subroutines to be shared with the xmit rework.

This process will start, however on the receive side with the introduction of a single MAST device. A member will added to the "ipsec_sa" structure which will point to an appropriate MAST device. This will be used to set the `skb->dev` pointer to on cleartext packets emerging. This differs from established practice of pointing it to the `ipsecX` device.

Note that the "ipsec_sa-¿odev" member will be initalized to the single MAST device at present as no method of setting it from above is provided at this stage.

The skb's fwmark will be filled in with the SAID index. As an optional measure, a new member will be added to the skb structure called the "flowpolicy". An attempt will be made to get this patch into the mainstream.

This will permit later code to be written to do the tunnel exit checks within NetFilter.

6.9 ipsec_tunnel_xmit rework

The transmit side will be reworked starting in October for a release in December.

The transmit work will include:

- adapt lifetime structure and checks.

- split into pre-crypto, crypto and post-crypto stages.

- use of low-level generic algorithm functions (e.g. 3DES, SHA1, MD5) to perform crypto.

- at the KLIPS level at least, stop overloading said SPI value usage. This should propogate back to Pluto via new PF_KEY extensions, but should remain compatible for a period.

- eliminate redundant lookup of eroute-¿TDB chain by linking eroute directly to ipsec_sa from eroute using reference counts.

- add facility for using netfilter for SA selection. If new "flowpolicy" field is set, then skip eroute lookup.
- connect MAST transmit to "default" SA. This means that any packets that are routed to the MAST device will have their flowpolicy set and be send to the IPsec transmit code.

6.10 netfilter extensions

There is one additional test needed by IPsec and one additional action. These are:

- a predicate to test flowpolicy for a value.
- an action to set flowpolicy to a value.

6.11 netfilter setup/usage by Pluto

There are four things which the setup scripts and/or Pluto will have to learn about. These are:

- management of multiple MAST devices
- netfilter code to implement SPD
- netfilter code to do exit checks
- extensions for eroute types in PF_KEY

Note that we are augmenting the current eroute code with netfilter that does the same thing at this stage. This permits experimentation with a new way to do things, while not impeding compatibility and bug fixes.

A future effort will expose an interface to the generic algorithm interface.