

SAHN Daemon Programming Guide

Mike Tyson

October 13, 2005

Contents

1	Programming practice	3
1.1	Messages and debugging	3
1.2	Comments	3
2	Events and callbacks	4
2.1	Packet handlers	4
3	Utilities	4
3.1	Packet buffers	4
3.1.1	Access	5
3.1.2	Manipulation	7
3.1.3	Allocation and deallocation	7
3.2	Lists	8
3.2.1	Declaring lists and list nodes	8
3.2.2	Modifying lists	8
3.2.3	Accessing lists	9

1 Programming practice

1.1 Messages and debugging

All messages should be written to the console via the following utilities:

DBG(level, action) Perform debug action.

If the debug level is set to ‘level’ or above, the given action is performed. Be aware that actions performed here will not occur when the debug level is below ‘level’.

DBGP(level, args) Print debug.

When debug level is above ‘level’, prints the given arguments. This utility takes arguments identical to *printf()*’s.

ERRP(args) Print error.

Prints the given arguments as an error. If debugging mode is enabled, will also report the current source file and line number. Treats arguments identically to *printf()*.

INFOP(args) Print information.

Prints given arguments as system log information. Treats arguments identically to *printf()*.

Debug printing should be done as much as possible, to aid debugging. Make use of the debug level parameters, which range from 0 (no debugging) to 9 (full debugging) The debug level is set via the command line ‘-debug’ parameter, and defaults to 0. Frequently performed actions should have debugging lines with a high debug level; less frequently performed actions should have a lower debugging level.

Comment/uncomment `DBG_SHOW_LINE_NUMBERS` in `common.h` to turn on/off line number printing for all messages.

1.2 Comments

For the sake of consistency, all functions should have Javadoc-style header blocks, which provide a brief description of what the function does, and a description of each parameter and any return values.

An example header comment:

```
/**
 * Grow packet tail
 *
 *      Grows the packet’s end downwards, and returns a pointer to
 *      the newly available space (the old end).
 *      Will not allocate any space if more space than available is
 *      requested; will return NULL in this case.
 *
 * @param pkt Packet
 * @param space Space to add
 * @returns Pointer to new space
 */
```

This allows for easy assessment of a function, and permits automatic generation of documentation with the aid of a tool such as Doxygen.

2 Events and callbacks

2.1 Packet handlers

Incoming packets are passed through an event handler which passes packets of a registered type to a registered function.

Such functions can be set with the *register_packet_handler* function, defined as:

```
int register_packet_handler(u8 type, packet_callback_fn callback);
```

The ‘type’ parameter represents the incoming packet type field value; Packets with a type of this value will be run through the given callback. Note that if several different handlers are registered for the same type, only one handler will be passed the packet. Care should be taken to avoid this scenario, as behaviour will be undefined.

Handlers should return SAHN_DROP to have the packet dropped, and SAHN_OK otherwise. Normally, it is appropriate to return SAHN_DROP, unless the packet is a broadcast and should be propagated.

3 Utilities

3.1 Packet buffers

The SAHN daemon and module make use of a structure to hold packet data and control information, partially modelled from Linux’s ‘sk_buff’ structure. This is defined in *sahn_pktbuf.h*, with comments.

Packet buffers are defined as:

```
struct sahn_pktbuf {
    struct sahn_devspec dev;           // Physical device name
    unsigned long space;              // Allocated space
    unsigned long head,               // Head offset
                 data,               // Data offset
                 tail,               // Tail offset
                 end;               // End of data
    unsigned long start_stack[SAHN_PKTBUF_STACKSIZE], // ‘Start’ offsets
                 end_stack[SAHN_PKTBUF_STACKSIZE];   // ‘End’ offsets
    short stack_ids[SAHN_PKTBUF_STACKSIZE];           // Stack item ids
    unsigned short stackp;                          // Stack pointer
    unsigned int user[3];                            // User fields
    void (*tx_fail_handler)(struct sahn_pktbuf*);    // TX failure handler callback
};
```

Packet buffers have data, tail space and head space; Head data grows upwards into the available headspace, while tail data grows downwards.

When created, *sahn_pktbufs* have zero header length, zero data length and zero tail length. To add header data, *sahn_pb_growhead* is used to prepare space. To add tail data, *sahn_pb_growtail* is used. Do not try to store headers/data without first preparing space.

A stack is used to delimit sections of packets – once a section is completed, *sahn_pb_push* is used to encapsulate the current packet data. The encapsulation is used to indicate the packet contents to other sections of the daemon, primarily the security/encryption module. See 3.1.2 for an example session.

For incoming data, encapsulation levels can be set with calls to *sahn_pb_encapsulate*. This is particularly necessary to allow the security/encryption module to identify the data to process.

There are three defined levels of encapsulation, which correspond to the three levels of packet data defined within the literature.

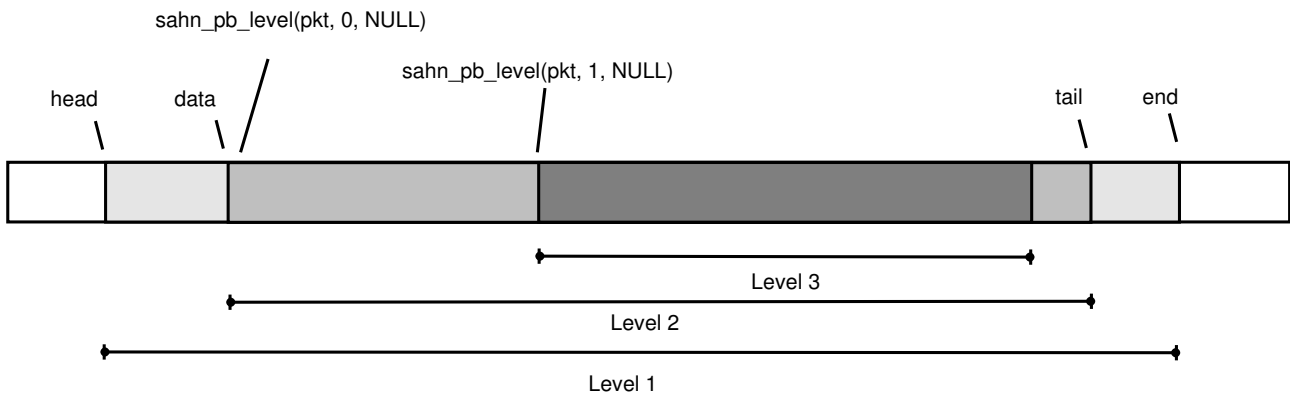


Figure 3.1.1: Example packet structure

3.1.1 Access

The 'head', 'data', and 'tail' methods should never be accessed directly. The following utilities exist for accessing packet buffer data:

sahn_pb_head(pb) Get header of packet

sahn_pb_tail(pb) Get tail of packet (space between data and end)

sahn_pb_alldata(pb) Get all data (headers & data)

sahn_pb_data(pb) Get data (contents of lower packet level)

sahn_pb_end(pb) Get end of packet (do not dereference)

sahn_pb_headlen(pb) Get length of head

sahn_pb_dataLen(pb) Get data length (length of lower packet level)

sahn_pb_tailLen(pb) Get length of tail

sahn_pb_len(pb) Get total length

sahn_pb_space(pb) Get total available space

sahn_pb_headspace(pb) Get available headspace

sahn_pb_tailSpace(pb) Get available dataspace

sahn_pb_level(pb,id) Get packet data at given level

sahn_pb_levelend(pb,id) Get end of packet data at given level

sahn_pb_levelLen(pb,id) Determine length of data at given level

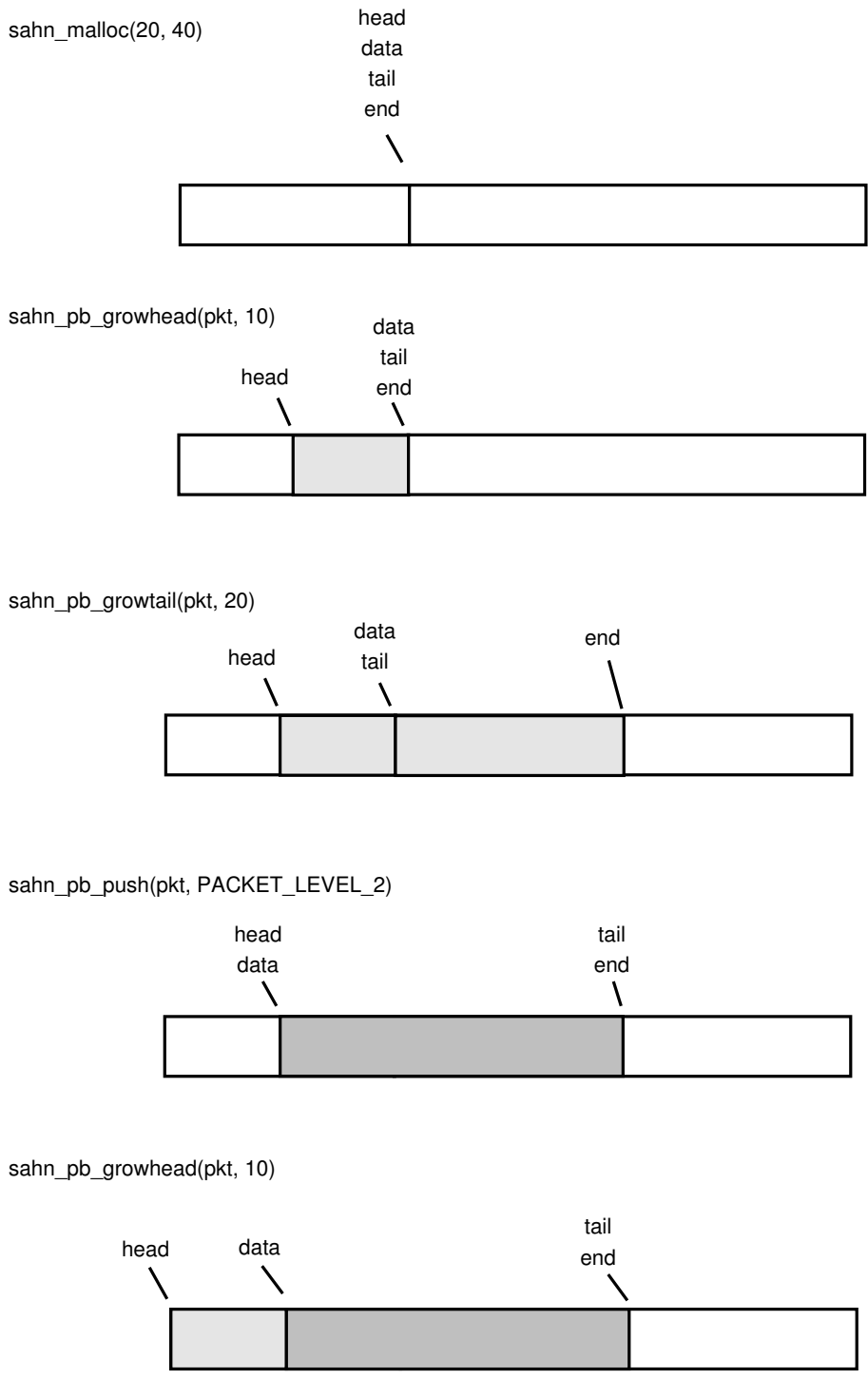


Figure 3.1.2: Example partial session

3.1.2 Manipulation

For manipulation of packet buffers, the following utilities should be used:

char* sahn_pb_growtail(struct sahn_pktbuf* pkt, unsigned int space) Increase tail size
Grows the tail area downwards, and returns a pointer to the newly available space. Will not allocate any space if more space than available is requested; will return NULL in this case.

Use this routine to append data to a packet.

char* sahn_pb_shrinktail(struct sahn_pktbuf* pkt, unsigned int space) Shrink tail
Shrinks the bottom edge of the tail area. If tail length is smaller than given figure, will just shrink by tail length.

char* sahn_pb_growhead(struct sahn_pktbuf* pkt, unsigned int space) Grow header
Grows header upwards into headspace. If given length is greater available headspace, operation will fail, and will return NULL.

Use this routine to increase header size

char* sahn_pb_shrinkhead(struct sahn_pktbuf* pkt, unsigned int space) Shrink header
Shrinks top edge of header area. If header length is less than given figure, header will be shrunk by header length.

char* sahn_pb_push(struct sahn_pktbuf* pkt, short id) Push current packet data
Encapsulates current packet data, including headers, within a level encapsulation.
Pushes identifier of current pointers onto a stack, allowing for later access via sahn_pb_level() and sahn_pb_levellen(). Afterwards, packet data points to previous packet head, packet tail points to end of packet, and head and tail lengths are zero.

If stack is full, will return 0.

char* sahn_pb_encapsulate(struct sahn_pktbuf* pkt, void* start, void* end, short id) Add packet encapsulation, accessible later with sahn_pb_level() and sahn_pb_levellen().

3.1.3 Allocation and deallocation

Routines exist to allocate and free packet buffers. These perform *ioctl()* calls which direct the SAHN module to perform allocation and free operations.

These routines behave much like *malloc()* and *free()*.

The routines are:

struct sahn_pktbuf* sahn_malloc(unsigned long headspace, unsigned long dataspace) Allocate space for a packet.

Allocates space for a single packet. Pass amount of required headspace, and required data space. Note that internally required space is automatically added to headspace. This is set by the *add_extra_default_headspace()* and *add_extra_default_tailpace()*.

If allocation failed, NULL is returned.

void sahn_free(struct sahn_pktbuf* pkt) Free packet buffer.

Frees given buffer; as in *free()*, the pointer should not be touched afterwards.

3.2 Lists

The SAHN daemon implementation contains linked list utilities that should be used when the need to store lists of items arises. Note that the implementation provides fast prepends, and fast deletes, but linear-time accesses. If fast accesses are required in moderately sized lists, an array or hash is a better option.

The list routines are located within `list.h`, and are implemented with a variety of macros, defines, and inline functions.

Lists can store any structure, as long as it is defined appropriately with a `LIST_ANCHOR` (see Section 3.2.1). The implementation can also store structures of different types within the same list.

3.2.1 Declaring lists and list nodes

Two routines exist for list declaration:

DECLARE_LIST Define list uninitialised.

This should be used inside structs or within functions. After declaring using this routine, the list should be initialised with `initialise_list(listname)`.

DECLARE_LIST_INIT Define a list and initialise.

This should be used when defining a static list, and includes initialisation. Note that this utility will not work within functions or structs.

Lists contain list nodes, which are structures defined with a `LIST_ANCHOR`. To define such a structure, insert the `DECLARE_LIST_ANCHOR()` declaration as the first item within the struct.

Example:

```
struct double_list {
    DECLARE_LIST_ANCHOR();
    double value;
};
```

Note that the `DECLARE_LIST_ANCHOR()` declaration must appear as the first item within the structure. Failure to meet this condition will result in undefined behaviour.

It is good practice to initialise list nodes before use. This can be achieved through the `initialise_list_node(name)` utility:

```
struct double_list* newnode =
    (struct double_list*) malloc(sizeof(struct double_list));
initialise_list_node(newnode);
newnode->value = 1.0;
```

3.2.2 Modifying lists

The following utilities exist for list manipulation:

list_prepend(node, list) Prepend to list.

Adds given node to beginning of list.

list_insert(node, prev) Insert into list.

Inserts given node after the given previous node.

list_append(node, list) Append to list.

Adds given node to end of list. This function is not recommended, as it runs in linear time, on the length of the list, as it must seek to the end of the list.

list_del(node) Delete from list.

Removes the given node from the list.

3.2.3 Accessing lists

The following utilities are available for accessing lists:

list_empty(list) Determine whether list is empty.

Will return 1 if the list is empty, 0 otherwise.

list_length(list) Get length of list.

Counts number of items in list. Note that this function runs in linear time on list length.

list_nth(list, n) Get nth item within list.

Seeks through list and returns nth item. This function is linear on n. If n is greater than the list length, NULL is returned.

list_first(list) Get first item in list.

Returns first item within list, or NULL if list is empty.

list_find(list, cursor, condition) Searches for item in list.

Given a list, a cursor of node type*, and a boolean expression using cursor, searches through list for matching items.

Example: *struct double_node* cursor; list_find(&list, cursor, cursor==1.0);*

Routines also exist for iterating through lists.

list_for_each(list, cursor) behaves like a *for()* expression, and will execute the following expression or block for each item within the list.

list_for_each_safe(list, cursor, tmp) provides similar functionality as above, but includes safeguards so that if the current node is deleted, the iteration will not be affected. Use this utility if expecting to delete nodes during iteration.

Example:

```
struct double_node *cursor, *tmp;
list_for_each_safe(&list, cursor, tmp)
{
    if ( cursor->value == 0.9 )
        list_del(cursor);
    else
        printf("%lf\n", cursor->value);
}
```