iptable reference

hertogp

Wrapper for radix.c by the FreeBSD project.

iptable.h

`#define's`

IPT_x

`IPT_KEYOFFSET` the offset to the actual key in the byte array
`IPT_KEYLEN(k)` the length of byte array `k`
`IPT_KEYPTR(k)` pointer to the location of the key in the byte array `k`

IP4_x

`IP4_KEYLEN` the length of the byte array to hold an IPv4 binary key
`IP4_MAXMASK` the maximum number of bits in an IPv4 binary mask
`IP4_PFXSTRLEN` buffer size for an ipv4 prefix string (addr/len)

IP6_x

`IP6_KEYLEN` the length of the byte array to hold an IPv6 binary key
`IP6_MAXMASK` the maximum number of bits in an IPv6 binary mask
`IP6_PFXSTRLEN` buffer size for an ipv6 prefix string (addr/len)

STR_x

`STR_IS_IP4(s)` detects an ipv4 address by an in-string '.'-character
`STR_IS_IP6(s)` detects an ipv6 address by an in-string ':'-character

KEY_x

`KEY_IS_IP4(k)` checks for an ipv4 binary key by checking the byte array's length
`KEY_IS_IP6(k)` checks for an ipv6 binary key by checking the byte array's length
`KEY_AF_FAM(k)` returns `AF_INET(6)` or `AF_UNSPEC` based on binary key length
`KEY_LEN_FAM(af)` returns byte array lengh for given AF family.

MAX_x

`MAX_BINKEY` buffer size to hold both ipv4/ipv6 binary keys
`MAX_STRKEY` buffer size to hold both ipv4/ipv6 prefix/len strings

AF_x

`AF_UNKNOWN(f)` true if f is not AF_INET or AF_INET6

RDX_x

`RDX_ISLEAF(rn)` true if radix node `rn` is a LEAF node
`RDX_ISINTERNAL(rn)` true if radix node `rn` is an INTERNAL node
`RDX_ISROOT(rn)` true if radix node `rn` is a ROOT node (LE-mark, TOP, RE-mark)
`MSK_ISROOT(rm)` true if radix mask node `rm` is a ROOT node (LE,TOP,RE)
`RDX_ISRCHILD(rn)` true if radix node `rn` is its parent right child
`RDX_MAX_KEYLEN` the maximum length for a byte array.

RDX FLAG

`IPTF_DELETE` additional radix node flag to indicate node was deleted

When an iteration is happening, radix nodes are flagged for deletion rather than being removed immediately. All iterators run with an upvalue that has a garbage collector which will eventually remove radix nodes flagged for deletion when it is safe to do so.

The flag is set in `rn_flags` in a radix node, alongside the flags defined by `radix.h`

RDX node types

A table has a stack which allows for pushing arbitrary data combined with a type identifier for that data. The stack is only used by the radix iterator which is read only and allows for graphing a radix tree. See 'ipt2dot.lua'

Radix node type include:

- `TRDX_NONE` – indicates an unknown type
- `TRDX_NODE_HEAD` – a radix_node_head
- `TRDX_HEAD` – a radix head, a struct inside a radix node head
- 'TRDX_NODE' – a single radix node
- `TRDX_MASK_HEAD` – the head of the mask tree
- `TRDX_MASK` – a radix mask node

Structures

`entry_t`

The type `entry_t` has 2 members:

- `rn[2]`, an array of two radix nodes: a leaf & an internal node.
- `void *value`, which points to user data.

The radix tree stores/retrieves pointers to `radix leaf nodes` using binary keys. So a user data structure must begin with an array of two radix nodes: a leaf node for storing the binary key, associated with the user data, and an internal node to actually insert the leaf node into the tree.

Retrieving the data is done by matching, either exact or using a longest prefix match, againts a binary key which yields a pointer to a leaf node. That pointer is then recast to `entry_t *` in order to access the user data associated with the matched binary key in the tree via the `value` pointer.

`purge_t`

The type `purge_t` has the following members:

- `struct radix_node_head *head`, the head of a radix tree
- `purge_f_t *purge`, a callback function pointer; to free user data
- `void *args`, an opague pointer to be interpreted by `purge`

This structure is used to relay contextual arguments to the user callback function upon deletion time. The different levels of memory ownership include:

- `radix.c`, which owns:
  - the 'mask' tree completely, but
  - only the radix_node_head for the 'key'-tree, not the leafs
- `iptable.c`, which owns:
  - the radix nodes that are part of `entry_t`, and
  - the binary key which it derives from `const char *` strings
- `user.c`, she owns:
  - the memory pointed to by the `void *value` pointer in `entry_t`

So when `user.c` calls `iptable.c`'s `tbl_create`, it supplies a *purge* function whose signature is:

```
typedef void purge_f_t(void *pargs, void *value);
```

When `user.c` calls `iptable.c`'s `tbl_del`, it supplies the prefix string to be deleted and a contextual argument for its `purge` callback function (`pargs` here). `tbl_del` then:

- derives a binary from the prefix given
- calls `radix.c's rnh_deladdr` to remove the binary key, (if succesful, two radix nodes are returned (ie an `entry_t`))
- frees the `entry_t`'s memory and the binary key memory, and finally

- calls back the `purge` function supplying it with both the
  - `void *pargs`, the contextual argument for this deletion, and
  - `void *value`, from the `entry_t` that was deleted.

It may seem a bit convoluted, but it allows the user's `purge` callback to examine a request to free user memory in context.

`stackElm_t`

A stack element has members:

- `int type`, denotes the type of this element
- `void *elm`, an opaque element pointer
- `struct stackElm_t *next`, pointer to the next stack element

An iptable contains a stack, which is only used by `rdx_firstnode` and `rdx_nextnode` to perform a preorder (node, left, right) traversal and node processing of all nodes of all types of all trees used in the table. I.e. it traverses both the 'key'-tree as well as the 'mask'-tree and yields all nodes of all types used in either tree.

This makes it possible to produce a dot-file of a radix tree and create an image using graphviz's dot tool.

`table_t`

An iptable has the following members:

- `struct radix_node_head *head4`, the head of the ipv4 radix tree
- `struct radix_node_head *head6`, the head of the ipv6 radix tree
- `size_t count4`, the number of ipv4 prefixes present in the ipv4 tree
- `size_t count6`, the number of ipv6 prefixes present in the ipv6 tree
- `purge_f_t *purge`, user callback for freeing user data
- `int itr_lock`, indicates the presence of active iterators
- `stackElm_t *top`, the stack to iterate across all radix nodes in all trees
- `size_t size`, the current size of the of the stack

Two separate radix trees are used to store ipv4 resp. ipv6 binary keys. Table operations detect the type of prefix used and access the corresponding tree. Each time a prefix is added or deleted, the tree's counter is updated. The `purge` function pointer is supplied upon tree creation time and is called whenever user data can be freed, see `purge_t`.

The `itr_lock` is actually a Lua specific feature to track the presence of any currently active tree iterators (there are a few). This allows for postponed radix node removal while some iterator is still traversing one of the trees.

Finally, the `*top` and `size` exist in order to be able to graph the tree(s).

iptable.c

max_mask

```
uint8_t max_mask[RDX_MAX_KEYLEN] = {-1, .., -1};
```

`max_mask` serves as an AF family's default mask in case one is needed, but not supplied. Thus missing masks are interpreted as host masks.

Key functions

key_alloc

```
uint8_t *key_alloc(int af);
```

Allocate space for a binary key of AF family type `af` and return its pointer. Supported `af` types include:

- `AF_INET`, for ipv4 protocol family
- `AF_INET6`, for the ipv6 protocol family

key_copy

```
uint8_t *key_copy(uint8_t *src);
```

Copies binary key `src` into newly allocated space and returns its pointer. The src key must have a valid first length byte. $0 <= KEY\_LEN(src) <= MAX\_KEYLEN$. Returns `NULL` on failure.

key_bystr

```
uint8_t *key_bystr(uint8_t *dst, int *mlen, int *af, const char *s);
```

Store string `s` binary key in `dst`. Returns NULL on failure. Also sets `mlen` and `af`. `mlen`=-1 when no mask was supplied. Assumes `dst`'s size MAX_BINKEY, which fits both ipv4/ipv6.

key_byfit

```
uint_8t *key_byfit(uint8_t *m, uint8_t *a, uint8_t *b)
```

Set `m` to the largest possible mask for key `a` such that:

- `a`'s network address is still `a` itself, and
- `a`'s broadcast address is less than, or equal to, `b` address

Note: the function assumes $a <= b$.

key_bylen

```
uint8_t *key_bylen(uint8_t *binkey, int mlen, int af);
```

Create a mask for given `af` family and mask length `mlen`.

key_bypair

```
uint8_t * key_bypair(uint8_t *a, const void *b, const void *m);
```

Set key a such that a/m and b/m are a pair that fit in key/m-1. Assumes masks are contiguous.

key_masklen

```
int key_masklen(void *key);
```

Count the number of consequtive 1-bits, starting with the msb first.

key_tostr

```
const char *key_tostr(char *dst, void *src);
```

src is byte array; 1st byte is usually total length of the array.

- iptable.c keys/masks are uint8_t *'s (unsigned)
- radix.c keys/masks are char *'s. (may be signed; sys dependent)
- radix.c keys/masks's KEYLEN may deviate: for masks it may indicate the total of non-zero bytes i/t array instead of its total length.

key_tostr_full

```
const char *key_tostr_full(char *dst, void *src);
```

Return the full string for a key without shorthanding contiguous zero's for ipv6 keys. If the src represents a mask, it may be shorter than than the protocol's actual key-length, so supply trailing zeros as well. See the remarks for `key_tostr`. Only has effect for ipv6 keys. Embedded ipv4 addresses are printed as hex digits, not as integers.

key_bynum

```
int key_bynum(void *key, size_t number);
```

Create key from number. Returns NULL on failure, key otherwise.

key_incr

```
uint8_t *key_incr(uint8_t *key, size_t num);
```

Increment `key` with `num`. Returns key on success, NULL on failure (e.g. when wrapping around the available address space) which usually means the resulting `key` value is meaningless.

key_decr

```
uint8_t *key_decr(uint8_t *key, size_t num);
```

Decrement `key` with `num`. Returns `key` on success, NULL on failure (e.g. when wrapping around the available address space) which usually means the resulting `key` value is meaningless.

key_invert

```
int key_invert(void *key);
```

inverts a key, usefull for a mask. Assumes the LEN-byte indicates how many bytes must be (and can be safely) inverted.

key_reverse

```
int key_reverse(void *key);
```

reverse the bytes of a key. For ipv6, the nibbles are also swapped. Assumes the LEN-byte indicates how many bytes must be (and can be safely) reversed.

key_network

```
int key_network(void *key, void *mask);
```

key_broadcast

```
int key_broadcast(void *key, void *mask);
```

set key to broadcast address, using mask

- 1 on success, 0 on failure
- mask LEN <= key LEN, 'missing' mask bytes are taken to be 0x00 (a radix tree artifact).

key_cmp

```
int key_cmp(void *a, void *b);
```

Returns -1 if a<b, 0 if a==b, 1 if a>b; or -2 on errors

key_isin

```
int key_isin(void *a, void *b, void *m);
```

return 1 iff a/m includes b, 0 otherwise note:

- also means b/m includes a
- any radix keys/masks may have short(er) KEYLEN's than usual

key_ynp

```c
uint8_t * key_ynp(uint8_t *d, uint8_t *s, int n, int off)

Yank `n` bytes from source key `s` and paste into destination `d` starting
at offset `off`.  Returns the resulting destination pointer (for subsequent
pasting) or NULL on failure.  Source key `s` should point to the LEN-byte of
the sourcing key.  If reading `n` bytes starting at `off` would read past
the end of the source key `s` LENgth, this function returns NULL.  It is
the caller's responsiblity to ensure `d` points into a buffer where it has
enough room left to receive `n`-bytes.  Note that offset `off` is
zero-based, starting at the LEN byte of `s`.  So to copy an entire key, call
`key_ynp(d, s, 0, (int)(*s));` which would copy the entire key.



### `key6_by4`
```c
  uint8_t *key6_by4(uint8_t *v6, uint8_t *v4, int compat)
```

Derive an ipv6-key from an ipv4 key:

- `::ffff:V4ADDR` if compat is 0, or
- `::V4ADDR` if compat is true

Returns 1 on success, 0 on failure (such as when v4 is not an IP4-key). Both v6 and v4 are assumed to be uint_8 buffers of at least MAX_BINKEY.

key6_6to4

```c
  uint_8t *key6_6to4(uint8_t *v6, uint8_t *v4)
```

Derive an ipv6-key from an ipv4 key:

- `2002:V4ADDR::`

Returns 1 on success, 0 on failure (such as when v4 is not an IP4-key). Both v6 and v4 are assumed to be uint_8 buffers of at least MAX_BINKEY.

key4_by6

```c
  uint_8t *key4_by6(uint8_t *v4, uint8_t *v6)
```

Derive an ipv4-key from the last 4 bytes of an ipv6 key: Returns 1 on success, 0 on failure (such as when v6 is not an IP6-key). Both v6 and v4 are assumed to be uint_8 buffers of at least MAX_BINKEY. Note: this does NOT check if the ipv6 key is `v4mapped` or `v4compat`, it just copies the last 4 bytes into the v4 buffer.
```

key_toredo

```
int key_toredo(int get, uint8_t *v6, uint8_t *ts, uint8_t *tc, int *udp,
               int *flags);
```

Toredo prefix is 2001:0000:/32, where the bits & byte lengths are:

- 32b: 0 - 31 = 2001:0000 the toredo prefix
- 32b: 32 - 63 = V4ADDR of toredo server
- 16b: 64 - 79 = flags CRAAAAUG AAAAAAAA (*)
- 16b: 80 - 95 = udp port (inverted)
- 32b: 96 -127 = V4ADDR of toredo clien (inverted) flags: C = 0 since rfc5991 (used to a client behind cone NAT) R = 0 (unassigned) U/G = 0/0 to emulate "Universal/local" / "Group/Individual" bits in MAC's? A = 0 or random since rfc5991.

If get is true, gets the details from the v6 key, otherwise it creates a new v6 key based on the toredo details provided. Returns 1 on success, 0 on failure.

radix node functions

__dumprn‘

```
void _dumprn(const char *s, struct radix_node *rn);
```

dump radix node characteristics to stderr

rdx_flush

```
int rdx_flush(struct radix_node *rn, void *args);
```

free user controlled resources.

Called by walktree, rdx_flush:

- frees the key and, if applicable,
- uses the purge function to allow user controlled resources to be freed. The purge function is supplied at tree creation time.

Note: rdx_flush is called from walktree and only on *LEAF* nodes, so the rn pointer is cast to pointer to entry_t. As a walktree_f_t, it always returns the value 0 to indicate success, otherwise the walkabout would stop.

rdx_firstleaf

```
struct radix_node *rdx_firstleaf(struct radix_head *rh);
```

Find first non-ROOT leaf of in the prefix or mask tree Return NULL if none found.

Note: the /0 mask is never stored in the mask tree even if stored explicitly using prefix/0. Hence, the /0 mask won't be found by this function.

rdx_nxtleaf

```
struct radix_node *rdx_nextleaf(struct radix_node *rn);
```

Given a radix_node (INTERNAL or LEAF), find the next leaf in the tree. Note that the caller must check the IPTF_DELETE flag herself. Reason is to allow upper logic to be performed across both deleted and normal leafs, such as actually deleting the flagged nodes..

pairleaf

```
struct radix_node *pairleaf(struct radix_node *oth);
```

Find and return the leaf whose key forms a pair with the given leaf node such that both fit in an enclosing supernet with the given leaf's masklength-1. Note:

- 0/0 is the 'ultimate' supernet which combines 0.0.0.0/1 and 128.0.0.0/1

rdx_firstnode

```
int rdx_firstnode(table_t *t, int af_fam);
```

initialize the stack with the radix head nodes

rdx_nextnode

```
int rdx_nextnode(table_t *t, int *type, void **ptr);
```

pop top and set type & node, push its progeny

- stackpush will ignore NULL pointers, so its safe to push those return 1 on success with type,ptr set return 0 on failure (eg stack exhausted or unknown type)

table functions

tbl_create

```
table_t *tbl_create(purge_f_t *fp):
```

Create a new iptable with 2 radix trees.

tbl_walk

```
int tbl_walk(table_t *t, walktree_f_t *f, void *fargs);
```

run f(args, leaf) on leafs in IPv4 tree and IPv6 tree

tbl_destroy

```
int tbl_destroy(table_t **t, void *pargs);
```

Destroy table, free all resources owned by table and user

- return 1 on success, 0 on failure

tbl_get

```
entry_t *tbl_get(table_t *t, const char *s);
```

Get an exact match for addr/mask prefix.

tbl_set

```
int tbl_set(table_t *t, const char *s, void *v, void *pargs);
```

tbl_del

```
int tbl_del(table_t *t, const char *s, void *pargs);
```

tbl_lpm

```
entry_t *tbl_lpm(table_t *t, const char *s);
```

tbl_lsm

```
struct radix_node *tbl_lsm(struct radix_node *rn);
```

Given a leaf, find a less specific leaf or fail

- used by tbl_lpm in case the match is flagged for deletion

tbl_stackpush

```
int tbl_stackpush(table_t *t, int type, void *elm);
```

tbl_stackpop

```
int tbl_stackpop(table_t *t);
```