NaCl

Not Another Configuration Language

Content

1. Datatypes **3**

2. Typed objects **4**

2.1 Iface 4

2.2 Gateway 6

2.3 Conntrack 8

2.4 Vlan 8

3. Untyped objects **10**

4. Functions **11**

4.1 Packet properties 12

4.1.1 IP properties 13

4.1.2 ICMP properties 13

4.1.3 UDP properties 13

4.1.4 TCP properties 14

4.1.5 CT properties 14

4.2 Functions inside functions 14

4.3 Referring to NaCl objects inside a function 15

1. Datatypes

Datatypes that exist in NaCl behind the scenes are:

- integer (a number, f.ex. 10 or (-10))
- IPv4 address (f.ex. 10.0.0.45)
- IPv4 cidr (f.ex. 10.0.0.0/24)
- bool (f.ex. false)
- string (f.ex. “Hi”)
- range (f.ex. 10-20 or 10.0.0.40-10.0.0.50)
- list (f.ex. [10, 20, 30])
- object (f.ex. { key1: 10, key2: 20 })

2. Typed objects

A typed object initialization has the following structure: <type> <name> <value>, where the type can be Iface, Gateway, Conntrack or Vlan.

2.1 Iface

An Iface is a type that has certain requirements. The following property must be specified for each Iface created:
- index (integer)

The following properties must be given if you don't set the config property to dhcp, or don't set the config property at all:
- address (IPv4 address)
- netmask (IPv4 address)

Other properties that can be specified are:
- gateway (IPv4 address)
- dns (IPv4 address)
- config (dhcp, dhcp-with-fallback or static)
- masquerade (can be set to true or false, where false is default)
- prerouting (names of functions)
- input (names of functions)
- output (names of functions)
- postrouting (names of functions)
- vlan (names of Vlans or anonymous Vlan objects)

The value of an Iface can be an object. The object consists of key value pairs, separated by comma, and the pairs are enclosed by curly brackets:

```
Iface eth0 {
        address:        10.0.0.45,
        netmask:        255.255.255.0,
        gateway:        10.0.0.1,
        dns:            8.8.8.8,
        index:          0
}
```

The value can also simply be the configuration type (config) you want the Iface to have: dhcp, dhcp-with-fallback or static. Different requirements are connected to each of these.

For example, if you only want to set an Iface configuration to dhcp, you can use this syntax:

```
Iface eth0 dhcp
```

But since the index property always has to be set, you also need to set this:

```
Iface eth0 dhcp
eth0.index: 0
```

The dhcp-with-fallback configuration requires you to specify a fallback address and netmask:

```
Iface eth0 {
        config: dhcp-with-fallback,
        index: 0,
        address: 10.0.0.45,
        netmask: 255.255.255.0
}
```

The static configuration also requires an address and a netmask to be specified. This is default, and doesn't need to be specified:

```
Iface eth0 {
        index: 0,
        address: 10.0.0.45,
        netmask: 255.255.255.0
}
```

An Iface's properties can be set outside an object specification as well. F.ex.:

```
Iface eth0 dhcp-with-fallback
eth0.index: 0
eth0.address: 10.0.0.45
eth0.netmask: 255.255.255.0
eth0.gateway: 10.0.0.1
```

These properties can be set anywhere in the NaCl file.

An Iface has 4 chain properties that functions can be pushed onto (we'll come back to functions later, but the name of a function can be set as an Iface's chain's value). These chains are prerouting, input, output and postrouting.

```
Iface eth0 dhcp
eth0.index: 0
eth0.prerouting: my_function
```

More than one function can be added to a chain, but only one function of the type Filter should be added to each. This is because a Filter always returns either drop or accept, and

the Filter that follows a Filter will never get called. If you want to add more than one function to a chain, you have to specify a list:

```
Iface eth0 {
        config: dhcp,
        index: 0,
        prerouting: [ my_filter, my_first_nat, my_second_nat ]
}
```

2.2 Gateway

A Gateway object mainly consists of routes. The value of a Gateway object can either be a list of route objects, or an object consisting of key value pairs, where each pair's value is a route object:

```
Gateway myGateway [
        {
                net: 10.0.0.0,
                netmask: 255.255.255.0,
                iface: eth0
        },
        {
                net: 10.10.10.0,
                netmask: 255.255.255.0,
                iface: eth1
        },
        {
                net: 0.0.0.0,
                netmask: 0.0.0.0,
                nexthop: 10.0.0.1,
                iface: eth0
        }
]
```

or

```
Gateway myGateway {
        route1: {
                net: 10.0.0.0,
                netmask: 255.255.255.0
        },
        route2: {
                net: 10.10.10.0,
                netmask: 255.255.255.0,
```

```
        iface: eth1
    },
    defaultRoute: {
        net: 0.0.0.0,
        netmask: 0.0.0.0,
        nexthop: 10.0.0.1,
        iface: eth0
    }
}
```

If you create a Gateway with named routes, you can refer to these routes elsewhere in the NaCl file to set values that you haven't already set inside the route:

myGateway.route1.iface: eth0

The possible properties of a Gateway route are:

- net (IPv4 address)
- netmask (IPv4 address)
- gateway (IPv4 address)
- iface (name of an Iface)
- nexthop (IPv4 address)
- cost (integer)

A Gateway can also contain other key value pairs than routes, but then the Gateway must be an object containing key value pairs.

Possible Gateway properties that can be set besides routes:

- send_time_exceeded (enable or disable your service's gateway to send ICMP time exceeded messages) (true or false)

```
Gateway myGateway {
    send_time_exceeded: true,
    route1: {
        net: 10.0.0.0,
        netmask: 255.255.255.0
    },
    route2: {
        net: 10.10.10.0,
        netmask: 255.255.255.0,
        iface: eth1
    },
    defaultRoute: {
        net: 0.0.0.0,
        netmask: 0.0.0.0,
        nexthop: 10.0.0.1,
        iface: eth0
```

```
        }
}
```

You can only create one Gateway object per NaCl.

2.3 Conntrack

You can only create one Conntrack object per NaCl. This represents the connection tracking object in your service. You don't need to specify a Conntrack object for it to exist in your service, you only need to specify it if you need to set any of its properties.

The following properties can be specified for the Conntrack object:

- limit (maximum number of connections) (integer)
- reserve (number of entries in the connection tracking map, where there are two entries per connection) (integer)

```
Conntrack myConntrack {
        limit: 20000,
        reserve: 10000
}
```

2.4 Vlan

The Vlan type is similar to the Iface object, but is meant to be added to an Iface's vlan property.

The following properties can be specified for a Vlan object:

- address (IPv4 address)
- netmask (IPv4 address)
- gateway (IPv4 address)
- index (integer)

Index, address and netmask are mandatory to specify.

```
Vlan myFirstVlan {
        index: 13,
        address: 10.50.0.10,
        netmask: 255.255.255.0
}
```

The Vlan can then be added to an Iface's vlan:

```
Iface eth0 dhcp
```

eth0.vlan: myFirstVlan

More than one Vlan can be added to an Iface's vlan:

```
Iface eth0 dhcp
eth0.vlan: [ myFirstVlan, mySecondVlan ]

Vlan mySecondVlan {
        index: 22,
        address: 10.60.0.10,
        netmask: 255.255.255.0
}
```

A Vlan object doesn't need to be created, however, to set an Iface's vlan property:

```
Iface eth0 {
        index: 0,
        address: 10.0.0.45,
        netmask: 255.255.255.0,
        gateway: 10.0.0.1,
        vlan: [
                {
                        index: 13,
                        address: 10.50.0.10,
                        netmask: 255.255.255.0
                },
                {
                        index: 22,
                        address: 10.60.0.20,
                        netmask: 255.255.255.0
                }
        ]
}
```

3. Untyped objects

You can create objects with values of any of the datatypes listed in section 1. The initialization of an untyped object has the following structure: <name>: <value>

myPort: 4040

myPorts: [30, 40, 50, 60]

myAddress: 10.0.0.45

myAddresses: [10.0.0.40, 10.0.0.50, 10.0.0.80-10.0.0.90, 30.20.10.0/24]

myCidr: 10.0.0.0/24

myCidrs: [10.0.0.0/24, 30.20.10.0/20, 100.20.32.50/32]

```
myObject: {
        key1: 10,
        key2: {
                key2-1: 50,
                key2-2: 60
        }
}
```

These objects can be used in your functions or as values to your Iface or Vlan properties, or to your Gateway routes' properties.

4. Functions

The initialization of a function has the structure: <type>::<subtype> <name> { <body> }

```
Filter::IP myIPFilter {
        if (ip.daddr == 10.0.0.45) {
                accept
        }

        drop
}

Filter::TCP myFilter {
        if (tcp.dport == 1500) {
                accept
        }

        drop
}

Nat::TCP myNat {
        if (tcp.dport == 1500) {
                dnat(10.0.0.50, 1500)
        }
}
```

The **type** is either Filter (if you want to create a firewall) or Nat (if you want to NAT any of the packets going through your network).

The **subtype** is either IP, ICMP, UDP or TCP. If you create an IP filter (Filter::IP), you only have access to check the properties of the IP part of the packet. However, since all packets are IP packets, you know that all packets will go through the filter.

If you create a TCP filter (Filter::TCP), you can check both IP and TCP properties, but only TCP packets will go through the filter. In the same way, if you create an UDP filter (Filter::UDP), you can check IP and UDP properties, and only UDP packets will pass through the filter. Same with ICMP (Filter::ICMP). Connection tracking (ct) properties can be checked in all filters.

The **body** of a function consists of if statements that results in a verdict or action.

Possible **actions** in **Filters**:

- drop (immediately drops the packet)
- accept (immediately accepts the packet)

- log (prints out the given string and/or the specified packet properties each time a packet reaches the action)

Possible **actions** in **Nats**:
- dnat (destination NATs the packet and returns)
- snat (source NATs the packet and returns)
- log (prints out the given string and/or the specified packet properties each time a packet reaches the action)

Drop, accept, dnat and snat are verdicts, and when a packet reaches a verdict, the function returns the verdict and the rest of the function is not executed for that packet. The log action is not a verdict in that way, it just prints the message that the user has specified if a packet gets to it. After that the function execution continues until a verdict is reached.

Examples of **drop actions**:
drop
drop()

Examples of **accept actions**:
accept
accept()

Examples of **log actions**:
log(“My log message\n”)
log(“The source address of the IP packet is “, ip.saddr, ”\n”)

Examples of **dnat actions**:
dnat(10.0.0.45)
dnat(8080)
dnat(10.0.0.45, 8080)

Examples of **snat actions**:
snat(10.0.0.45)
snat(8080)
snat(10.0.0.45, 8080)

4.1 Packet properties

The conditions in an if statement can test on packet properties and you can use ‘and’ and ‘or’ between the conditions:

```
Filter::TCP myTCPFilter {
        if ((ip.daddr == 10.0.0.45 or ip.daddr == 10.0.0.50) and tcp.dport == 8080) {
                log(“Accepting packet with destination address”, ip.daddr, ”\n”)
```

```
            accept
        }

        drop
}
```

4.1.1 IP properties

- version (IP version) (integer)
- hdrlength (header length) (integer)
- dscp (differentiated services code point) (integer)
- ecn (explicit congestion notification) (integer)
- length (the total length of the packet in bytes) (integer)
- id (identification number) (integer)
- frag-off (fragment offset) (integer)
- ttl (time to live) (integer)
- protocol (protocol used in the data portion of the IP datagram) (ip, icmp, udp, tcp)
- checksum (header checksum, used for error-checking) (integer)
- saddr (source address) (IPv4 address)
- daddr (destination address) (IPv4 address)

4.1.2 ICMP properties

- type (type of ICMP message) (echo-reply, destination-unreachable, redirect, echo-request, time-exceeded, parameter-problem, timestamp-request, timestamp-reply)

Example condition in an ICMP Filter:

```
if (icmp.type == destination-unreachable) {
        drop
}
```

4.1.3 UDP properties

- sport (source port) (integer)
- dport (destination port) (integer)
- length (length of the UDP header and data in bytes) (integer)
- checksum (header checksum, used for error-checking) (integer)

4.1.4 TCP properties

- sport (source port) (integer)
- dport (destination port) (integer)
- sequence (sequence number) (integer)
- ackseq (acknowledgement number) (integer)
- doff (data offset) (integer)
- reserved (reserved for future use, should be zero) (integer)
- flags (contains 9 1-bit flags) (integer)
    - ns (ECN-nonce, nonce sum)
    - cwr (congestion window reduced)
    - ece (ECN-Echo)
    - urg (urgent pointer field is significant or not)
    - ack (acknowledgment field is significant or not)
    - psh (push)
    - rst (reset the connection)
    - syn (synchronize sequence numbers)
    - fin (last packet from sender)

  Future functionality: if (tcp.flags != syn) { drop }
- window (size of the receive window (number of window size units)) (integer)
- checksum (header checksum, used for error-checking) (integer)
- urgptr (urgent pointer) (integer)

4.1.5 CT properties

- state (connection tracking state) (established, new, invalid)

4.2 Functions inside functions

You can also have functions inside functions, but this is probably only desired inside an IP function, since a TCP function f.ex. already has access to the IP properties. The functions that are specified inside a function don't need to be named. You cannot mix different types of functions when doing this, you can only have Filters inside Filters and Nats inside Nats.

```
Filter::IP myFilter {
        if (ct.state == established) {
                accept
        }

        Filter::ICMP {
                if (icmp.type ==  echo-request) {
                        accept
```

```
                }

                drop
        }

        Filter::UDP {
                if (udp.dport == 60) {
                        accept
                }

                drop
        }

        Filter::TCP {
                if (tcp.dport == 80) {
                        accept
                }
        }

        drop
}
```

4.3 Referring to NaCl objects inside a function

As previously mentioned, you can create untyped and typed objects in your NaCl file and refer to them inside a function.

```
Iface eth0 {
        index: 0,
        address: 10.0.0.11,
        netmask: 255.255.255.0,
        gateway: 10.0.0.1,
        input: myFilter
}

myAddrs: [ 10.0.0.40-10.0.0.50, 120.0.10.0/24, 110.20.30.17 ]
myPorts: [ 8080, 9090, 1000-1200 ]

Filter::IP myFilter {
        if (ip.daddr in myAddrs or ip.daddr == eth0.address) {
                accept
        }
```

```
        Filter::TCP {
                if (tcp.dport in myPorts) {
                        accept
                }
        }

        drop
}
```