Advanced VPN Support on FreeBSD Systems

Advanced VPN support on FreeBSD systems

Riccardo Scandariato, Fulvio Risso
Politecnico di Torino, Italy
{scandariato, risso}@polito.it

Abstract— **Currently, the Virtual Private Network (VPN) support offered by FreeBSD is quite limited: it provides a way to establish tunnels but it does not consider the problems of multiple VPNs concurrently deployed on the same machine. Our implementation enables the provisioning of VPN services on FreeBSD by extending its routing and forwarding infrastructure. We adopted the virtual router approach, by adding support for multiple routing tables. Forwarding kernel modules have also been modified accordingly. We also improved several user-level applications (e.g. route, ifconfig, zebra) to allow the exploitation of the new routing infrastructure.**

Keywords— **Provisioned IP VPN, virtual router, GRE tunnel, FreeBSD**

I. Introduction

THE Internet, originally born as an academic-based infrastructure, is rapidly evolving toward a generic network in which academics, business, and several other worlds are coexisting. From the pure networking perspective (i.e. we do not intend to take into account any application issue), one of the problems of the nowadays public IP networks is the lack of support for IP private addresses. At a glance, supporting a private addressing schema on a public IP network seems to be a non-sense. However, from the perspective of companies with a wide area network infrastructure, this is a strong requirement since the IP public network is becoming a way to connect together their branch networks around the world (and saving money). This is the well-know topic under the name of Virtual Private Networks (VPN), i.e. networks that use a public IP infrastructure to connect together several pieces with private addressing (and with the need of secured communications). The biggest issue in VPN support is that the IP protocol did not foresee the need of multiple overlapping addresses spaces, so that applications like VPNs introduce a high degree of complexity in the management of the IP network. The idea of a VPN is not a novelty in the networking world. The novelty is that, so far, companies created their private network by using a data-link infrastructure (for example point to point links, or X.25 / Frame Relay / ATM accesses) provided by a telecom provider. That infrastructure was simply a private network (i.e. a network that allowed only the employees to access to the resources of the company) build on top of a public network (the public telephone network instead of the public X.25 network or whatever). In other words, the basic technology changes while the underlying concept of VPNs does not.

In order to support efficiently VPN services, there are two main options:

- the presence of the VPN is relegated to the access side of the network. This means that any VPN is hidden in the core (i.e. in the public part of the IP network) and the complexity is inserted into the edge routers. These routers must map the private address space into a public space, deliver packets to the proper destination, and then map the packets back.
- the presence of the VPN is well known inside all the network (backbone included), so that the routers must be aware that overlapped address spaces exist and they must be able to cope to this problem.

The first option is far simpler, but (in general) it does not allow creating an optimized virtual network from the topological viewpoint. The second option is more complex, but the routing into the network can be highly optimized. The present solutions (MPLS, tunneling, ...) chose the first method to deal with these problems. In our mind, however, these solutions cannot be used to configure plug and play networks, nor can be used to create large virtual infrastructures.

This work wants to present the lessons learned by modifying the forwarding path of a software router (FreeBSD 4.4) in order to support VPNs in the network backbone. Our effort was devoted to change the forwarding mechanism in order to support overlapped address spaces. Each router is then able to find the proper route to each packet by checking at the destination address (contained into the packet) and an additional parameter identifying the VPN the packet belongs to. However, this choice implies several other components to be modified in order to support the VPN into the backbone. Routing protocols, for example, need to be modified in order to be aware of what VPN they are currently computing the best path. Therefore, several other components (detailed in the following) have been modified in order to provide seamless VPN support.

The rest of the paper is organized as follows. Section II introduces the functionalities that are needed for concurrent VPN service provisioning. Section III describes the modification introduced into the FreeBSD kernel and into some user-space applications in order to implement such functionalities. Section IV discusses the related work, and, finally, the conclusive remarks and further work are presented in Section V.

II. Objective

As mentioned in Section I, VPN services can be implemented either on top of access routers only, or cooperatively afforded by all backbone routers. In the first case ([1], [2]), VPN traffic is identified when entering the backbone network, and then delivered to the opposite network edge by means of tunnels. Core routers are unaware of VPNs since they forward VPN tunneled traffic by looking up the destination address of the outer IP header (see later). All the VPN-specific work is done by access routers sitting at the backbone edge. For instance, routing information about VPN destination is exchanged only between access nodes. On the other case, VPN traffic is tagged at access node and sent through the core without encapsulation. All the core nodes forward packets using the couple (destination address, VPN_ID tag). This means that routing information about VPN destinations must be available to all the backbone nodes,

which have to maintain separate routing table (one for each tag, i.e one for each VPN).

In the first case, VPN packets are forwarded edge-to-edge across tunnels, which traverses multiple physical link (see tunnels between `freebsd` and `dante` traversing a third node in Figure 1). Obviously, since core nodes (e.g. `core001` in Figure 1) look up packets by using only the outer destination address, VPN packets (and tunnels) follows the physical paths governed by the "real" IP network (hereafter called the *base* network, in opposition to *virtual* networks). In other words, a single hop on the virtual network implies multiple hops on the base networks. This causes many problems if traffic engineering or QoS guarantees must be applied to tunnels, since the routing instances of the two types of networks (base and virtual ones) are unrelated. However, this solution simplifies the management of the core side.

The second case is logically equivalent to a network where all the backbone nodes (both access and core) play the role of VPN routers, and where tunnels are established on each physical link. Hence, a single hop on the virtual network coincides with an hop on the base network [1]. Hence, all the conventional techniques for traffic engineering and QoS can be applied to VPNs. The drawback is a huge complexity in the core side.

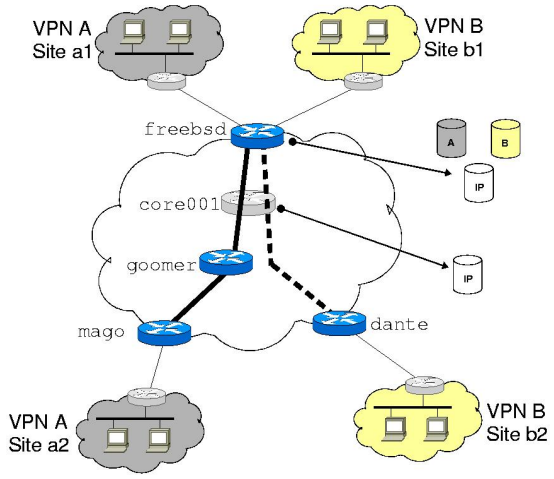

Fig. 1. Backbone network provisioning multiple VPNs

Our solution tries to merge the benefits of both the above approaches. Figure 1 shows a sample scenario that can be deployed by adopting our implementation. The figure depicts four customer sites connected to a provider network (the bigger cloud in the middle). The backbone is made of VPN routers (see darker nodes) and VPN-unaware routers (e.g. node `core001`). All the nodes providing access to client sites (`freebsd`, `mago`, `dante`) must be VPN nodes. Further, also some nodes in the core (`goomer`) can be promoted to be VPN router. VPN routers run the modified version of FreeBSD providing multiple table support and virtualized forwarding, as described in Section III. Having VPN routers in the core gives an increased flexibility for VPN deployment. For instance, a central node can be used as tunnel concentrator, in order to apply traffic filters to a given VPN, or to merge traffic originated in different VPNs. We recall that packet are available in clear, i.e. not tunneled, only at tunnel termination points, hence, such operation can be performed only by a VPN router. In Figure 1, a pure edge-based adopted was adopted for `VPN B` (see the dashed edge-to-edge tunnel between access nodes labeled as `freebsd` and `dante`), while `VPN A` uses an intermediate VPN node for traffic delivery (see the couple of tick tunnels between `freebsd`, `goomer`, and `mago`). Note that all VPN routers are maintaining a dedicated routing table for each VPN they are serving, additionally to the base network IP table. Hence, `freebsd` has to maintain 2+1 tables, while other VPN nodes are maintaining 1+1 tables. Non-VPN nodes just maintain the IP base table.

As highlighted by Figure 3 an Figure 2, access and core nodes fulfill different tasks. This are detailed in next two sections.

A. Access router functionalities

Figure 2 shows the detail of the `freebsd` VPN router. It is placed at the network edge and it has three physical interfaces:

- `eth0` is an Ethernet interface connecting client site VPN_A.a1 (site a1 of VPN A). This site is using the 10.0.1.0/24 address space.
- `eth1` is a second Ethernet interface connecting client site VPN_B.b1. This site is using the 10.0.1.0/24 address space too.
- `eth2` attached to the backbone core and has a public IP address.

Bounded to eth2, there are two pseudo-interfaces (also called virtual interfaces), that can be created dynamically:

- `gif0` is a GRE (Generic Routing Encapsulation, [3]) interface. This interface represents the tunnel end-point used to forward traffic of VPN A. The GRE interface is assigned (by means of the `ifconfig` UNIX command) with a private address out from the VPN A address space.
- `gif1` is another GRE interface. This interface represents the tunnel end-point used to forward traffic of VPN B. The GRE interface is assigned with a private address out from the VPN B address space.

The binding between virtual interface and the physical interface is done by means of the `gifconfig` BSD command, which configures the source and destination addresses to be used in the outer header.

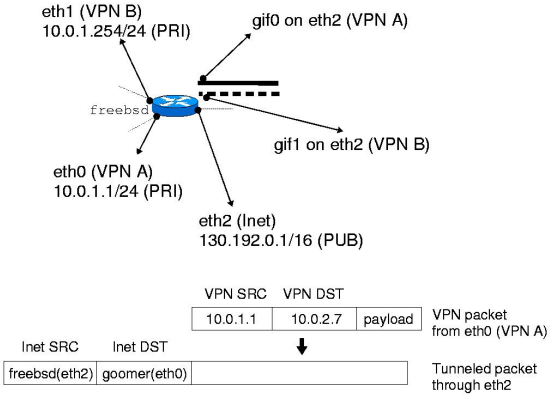

Fig. 2. VPN access router managing multiple VPNs

[1] Actually, while in the first case networks are layered (being virtual networks on top of the base networks) in the second case networks are sided (being the base network just one of the existing parallel networks).

In order to properly serve the two VPN client sites, the node must implement the following functionalities:

- *Routing virtualization*. The node must have multiple IP routing tables (one for each VPN) in order to support the overlapping address spaces within the two different VPNs. Each routing table contains path informations about all the other VPN destinations. Each route contains the address of a peer tunnel end-point (i.e. a `gif` on a remote tunnel-connected VPN router) as next hop. In our solution, each routing table is updated by a dedicated routing protocol instance (`zebra`-OSPF) running on top of the virtual network. This means that routing advertisement are tunneled too.
- *Incoming traffic identification*. The node must associate each packet incoming from the `eth0` and `eth1` interfaces to the corresponding VPN. To this aim, our solution "colors" the ingress physical interface by tagging them with a VPN identifier. This solution is straightforward to implement. However it imposes some limitations, as described in Section V.
- *Forwarding virtualization*. Upon reception of a packet from the client site, the forwarding module must be able to select the proper routing table according to the identified VPN. After having looked up the correct table, the forwarder must sent out the packet on one of the tunnels that have been configured for the given VPN. Selection of the right tunnel is determined by the VPN-level routing information.
- *Tunneling*. As shown in the lower part of Figure 2, once an outgoing tunnel has been selected, the corresponding `gif` module is responsible of encapsulating (i.e. of adding the outer header) the packet before transmission. This functionality (differently from all the above items) is already available in FreeBSD, and was not implemented.

Note that above we explained the case of a packet arriving from the client site. Obviously, the same operations must be pursued in case of delivery to the site of a packet arriving from the core.

B. Core node functionalities

Figure 3 shows the detail of the `goomer` VPN router. It is placed in the network core and it has two physical interfaces connecting him to other routers of the backbone. These interfaces are assigned with addresses out from the provider public address space.

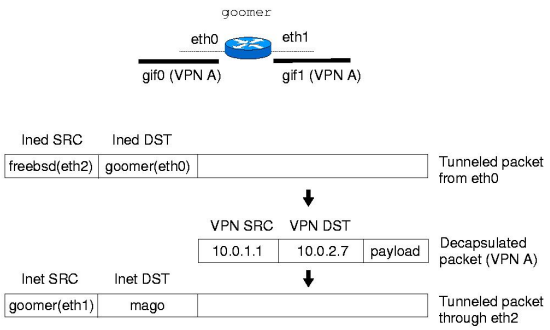

Fig. 3. VPN core router acting as a tunnel switch

The router also has two tunnel end-points, both terminating tunnels that belong to the VPN A:

- `gif0` represents the tunnel end-point used to forward traffic of VPN A to/from `freebsd`. The GRE interface is assigned with a private address out from the VPN A address space.
- `gif1` represents the tunnel end-point used to forward traffic of VPN A to/from `mago`. This interface is assigned with a private address out from the VPN A address space too.

Note that `gif0` is bounded to `eth0`, while `gif1` is bounded to `eth1`.

The node acts as a tunnel switch, that is it receives VPN traffic from a (incoming) tunnel and forwards it to another (outgoing tunnel). This means that the node has to decapsulate the incoming VPN packet, looking up the destination of the inner header towards the proper table, and then encapsulating the packet again (but with a new outer header). This operation is described in the lower part of the Figure 3.

With respect to the access case, the difference is in the traffic identification operation that is much more simpler, since the VPN traffic already arrives in a tunneled manner. Hence, a tag applied to the tunnel (actually the tunnel end-point) will serve the aim perfectly. Differently from Section II-A, this solutions does not involve any limitation, since end-point can be created dynamically in any desired number (up to the kernel configured limit). The only problem (as shown in Section V) is that the current tunneling implementation does not support more than one configured tunnel between a couple of IP addresses.

III. Implementation

This section outlines the strategy adopted to integrate the VPN functionalities described in Sections II-B and II-A into the FreeBSD 4.4 operating system. Since we refer frequently to the internals of FreeBSD, the reader unfamiliar with the networking architecture of a BSD-like system can refer to [4] and [5]. The detailed description of all the kernel/application modification can be found in [6], or directly in the source code [7].

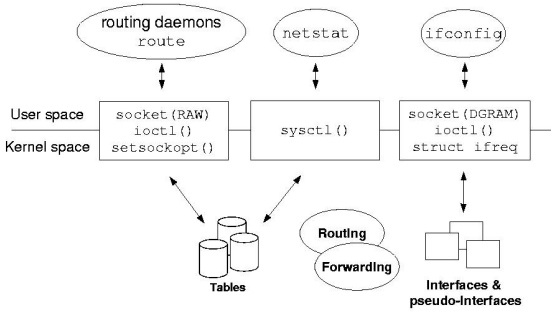

Fig. 4. Roadmap of VPN functionalities

Figure 4 gives the roadmap of modifications introduced by our implementation. Actually, the total amount of modified lines of code is very small, since our aim was to implement our solution in the most simple and clean way. Further, we tried to realize the most harmonic solution, with respect with the existing (i.e. original) FreeBSD code. Improvements were introduced both in the kernel space, and to application programs in the user space. Consequently to the introduction of new features in the kernel, we also modified the interface provided by some system calls, in order to provide applications with the new kernel

capabilities.

At the kernel level, the most important modification introduced the support for the on-demand creation of multiple routing tables, and for the tagging of interfaces. This features are exported by modified versions of the `socket()` and `ioctl()` system calls (an the modified versions of the related structures and ancillary functions). These features are exploited by utility programs (such as `route` and `ifconfig` respectively), and routing applications (such as `zebra-ospfd`, [8]), that were both modified. We also upgraded the `sysctl()` system call, providing bulk access to the routing table. This call is mainly used by the `netstat` program to get all the routing table at once. Hence we modified `netstat`, which is now capable of accessing the different tables. Obviously, the main part of the work was dedicated to the modification of routing mechanisms (table management) and forwarding functions (`ip_input()`, `ip_forward()`) in the kernel space.

The following sections detail the introduced variants to the FreeBSD system. In particular, Sections III-A and III-B present the improvements to the routing and forwarding modules respectively, while Section III-C presents the improvements we made to the user-space routing applications.

A. Multiple tables

FreeBSD supports many network protocols, such as IPv4, IPv6, IPX, etc. Since each one uses a single (and peculiar) addressing scheme, the protocol is internally identified by means of its *address family* number. For instance, the constant `AF_INET` identifies the IPv4 protocol, while the constant `AF_OSI` identifies the OSI protocol. The kernel assigns a dedicated routing table to each protocol (family), and tables are implemented as Patricia's trees, which are data structures optimized for longest-prefix-match searches. Tables are stored in a the `rt_tables[AF_MAX+1]` array, where `AF_MAX` is a constant representing the number of defined address families (i.e. the number of supported transport protocol). Each element of the array contains a pointer to the radix node of the corresponding tree-based table: for instance `rt_tables[AF_INET]` points to the IPv4 routing table. Tables are created and initialized at system startup by the `route_init()` function, which iteratively calls the `rn_inithead()` function, once for each family.

In order to support multiple routing tables for the `AF_INET` family, we defined an additional array as follows

```
(sys/socket.h) #define VPN_MAX 100
(net/route.h)  struct radix_node_head *
               vpn_rt_tables[VPN_MAX+1];
```

The maximum number of tables (and hence VPNs) is is limited by the `VPN_MAX` constant, since the array is statically allocated. To support a higher number of VPNs, the kernel must be recompiled with a different constant value. This choice is due to efficiency reasons. Further, the first element is not used, since the zero value is reserved to identify the base IP table. This design trick considerably reduces the number of modifications to the original kernel code (as explained later). Note that table structures are not created at startup time as above: initially, the vpn table array is empty, and Patricia's trees are created and initialized on-demand when a new VPN must be supported by the local node.

Once the multiple table support was introduced, we modified the interface providing the read/write access to the tables. User space programs communicate with kernel functions that manage the routing tables by means of *routing messages* exchanged through *routing sockets*. Routing messages are data structures defined in kernel headers that the programs fill in accordingly to the operation they want to execute on the table (e.g. route add, route delete, read, etc.). Routing sockets are created through the standard `socket()` system call, by specifying proper arguments. To allow the selection of the target table to which the operations must be executed, we introduced a new field (`so_vpnid`) in the `socket{}` data structure, as shown in Figure 5.

```
(sys/socketvar.h)
struct socket {
  short   so_state;  /* internal state */
  caddr_t so_pcb;    /* control block */
  ...
  u_int   so_vpnid;  /* VPN_ID -  ADDED */
}
```

Fig. 5. Socket data structure with VPN identifier

The `so_vpnid` field is initialized to zero by the `socreate()` function, when a new socket is requested through the `socket()` call. If the field is unmodified, all the routing messages will affect the base IP table (recall that VPN 0 is reserved). Thus, to select a table, the application program must set the `so_vpnid` field to a non-zero value, corresponding to the desired target table. To this aim, a modified version of the `ioctl()` system call is provided, which accepts the new `SIOCSVPNID` (set) and `SIOCGVPNID` (get) arguments. Alternatively, the VPN_ID can be set/read by means of a modified version of the `setsockopt()/getsockopt()` calls, respectively (which accept the new `SO_VPNID` socket level option). A sample code showing the use of the modified socket interface is provided in Figure 6.

At the kernel level, we modified the functions that process routing messages. Messages are first received by the `route_output()` routine. In case of read requests (`RTM_GET` is specified in message headers, similar to line 7 of Figure 6), it calls the `rnh_lookup()` function; otherwise, if the message requests a table modification (`RTM_ADD`, `RTM_DELETE`), it calls the `rtrequest()` routine. This latter, selects the target table on the basis of the address family of the route to be added or deleted.

We modified the default behavior of the `route_output()` function. Since, the `rtrequest()` cannot infer the VPN_ID from its input arguments, we were obliged to redefine the function as `vpn_rtrequest()`, which receives the VPN_ID as its last arguments. If the message is directed to the base table, this argument is zero. On the contrary, the `route_output()` function receives (as input argument) the pointer to the socket that transmitted the routing messages. Hence, it can extract the VPN_ID from the socket structure and can pass it to the `vpn_rtrequest()`. This latter selects the proper table corresponding to the received

```
1.  unsigned int vpnid = 5;
2.
3.  struct {
4.    struct  rt_msghdr header;
5.    char    body[512];
6.  } msg;
7.  msg.header.rtm_type = RTM_ADD;
8.
9.  int s = socket(PF_ROUTE, SOCK_RAW, 0);
10. ioctl(s, SIOCSVPNID, &vpnid);
11. // Alternatively ...
12. // setsockopt(s, SOL_SOCKET, SO_VPNID,
13. //            &vpnid, sizeof(vpnid));
14.
15. write(s, (char *)&msg, sizeof(msg));
```

Fig. 6. Sample code adding a route to VPN table no. 5

VPN_ID and executes the requested operation. Note that, if the table does not exists yet, the function dynamically creates and initializes a new Patricia's tree. To limits the number of modified line of code in function redefinition, we used a macro as shown in Figure 7. The adoption of macro redefinition, together with the association of the VPN zero to the base network, made modifications simpler and clearer. This strategy was used extensively throughout the code.

```
(bar.h)
1. // void foo(int, int);
2. void vpn_foo (int, int, u_int);
3. #define  foo(a, b) (vpn_foo(a, b, 0))

(bar.c)
1. void
2. //foo(a, b)
3. vpn_foo(a, b, vpnid)
4.   int a;
5.   int b;
6.   u_int vpnid;
7. {
8.   ...
9. }
```

Fig. 7. Example of code modification

Routing messages can also be generated by the kernel itself upward the applications, e.g. when a network interface goes down. Another example is the static configuration of a route through the `route` command. In this case, the kernel is responsible for the notification of the table update event to the routing daemons. These messages are sent to all the applications that have an open routing socket for the same transport protocol of the modified table. Applications specify the protocol they are interested in via the third argument of the `socket()` system call (zero means all protocols). The upward messages are processed by the `route_output()` routine, which in turn calls the `raw_input()` function. As above, we redefined this latter as `vpn_raw_input()`, which dispatches the messages according to both the protocol and the VPN_ID of open routing sockets.

B. Forwarding virtualization

Figure 8 sketches the kernel functions processing IP packets during forwarding. When a network interface receives an IP packet, it places the packet in the input queue. When the `ip_input()` function is scheduled, it fetches a packet from the queue head and processes it. For instance, the function analyzes the presence of eventual IP options, and if the packet is directed to a non local destination, it is passed to the `ip_forward()` routine for delivery. Packets are stored in a data structure called `struct mbuf` that contains both the packet data and related information, such as the ingress interface that received the packet. If the packet is tunneled, `ip_input()` passes the `mbuf` to the `gif_input()` routine, which decapsulates the packet (i.e. it strips the outer header off) and replaces the ingress interface in `mbuf` by putting the proper `gif` interface that terminates the tunnel, in place of the physical interface (e.g. `eth0`). Finally, `gif_input()` queues the packet again. Next time, the `ip_input()` will pass the decapsulated packet directly to the `ip_forward()`, and the `mbuf` will point to the `gif` ingress interface. The `mbuf` is available to both the `ip_input` and the `ip_forward()` routines as input argument, hence both can obtain a pointer to the ingress interface (being it a physical interface for access VPN router, a virtual one for VPN core nodes).

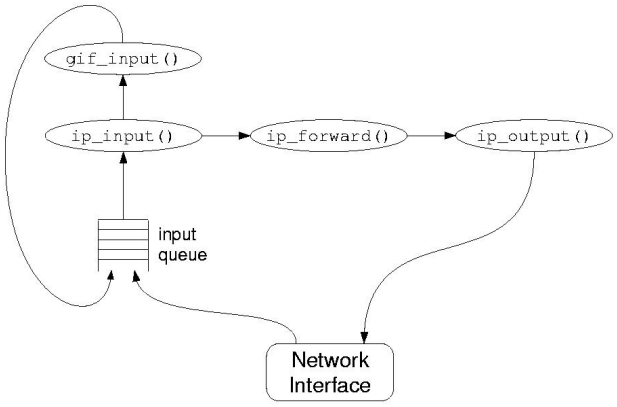

Fig. 8. Forwarding module in FreeBSD

The `ip_forward()` invokes the `rtalloc_ign()` to lookup the (base) routing table, and then sends the packet to the proper output interface. Transmission is mediated by the `ip_output()` that, for example, decrements the packet TTL and finally invokes the interface driver transmission routine.

To virtualize the forwarding process, we modified the default behavior of the `ip_forward()` routine. First, we added the VPN_ID to the interfaces by inserting an additional field to the `ifnet{}` data structure, as illustrated in Figure 9. We instructed the `if_attach()` routine (which is called to initialize all the interfaces, even for the dynamically created ones) to set the `if_vpn` field to zero. We modified the `ioctl()` system

```
(net/if_var.h)
struct ifnet {
  char    *ifname;   /* name, e.g. eth, gif */
  u_short if_index   /* numeric abbreviation */
  ...
  u_int   if_vpnid;  /* VPN_ID -  ADDED */
}
```

Fig. 9. Interface data structure with VPN identifier

call (and the `ifioctl()` ancillary function), which now accepts the `SIOCSIFVPNID` and `SIOCGIFVPNID` parameters to set/get the interface VPN_ID field. The modified version of `ifconfig` (which now accepts the `vpnid` switch), uses this system call, as shown in Figure 10.

```
1.  struct ifreq ifr;
2.  int    s, vpnid;
3.  char   ifname[16] = "eth0";
4.
5.  s = socket (AF_INET, SOCK_DGRAM, 0);
6.  vpnid = 5;
7.
8.  /* specify interface */
9.  strcpy(ifr.ifr_name, argv[2]);
10.
11. /* set interface VPN-ID */
12. ifr.ifr_vpnid = vpnid;
13.
14. ioctl(s, SIOCSIFVPNID, (caddr_t)&ifr);
```

Fig. 10. Code sample in `ifconfig` to set the VPN_ID on `eth0`

The `ip_forward()` was modified in order to call the `vpn_rtalloc_ign()` if the VPN_ID of the ingress interface is set to a non-zero value. The `vpn_rtalloc_ign()` (and the ancillary functions) is a redefined version of the standard `rtalloc_ign()`, which selects the correct table by using the `if_vpnid` field before looking up for the next hop. As a final result, packet forwarding is done by jointly considering both the destination address and the VPN_ID.

C. Routing daemons

This point can be seen as less important compared to the previous ones because it does not involve the modification of the operating system. Indeed, it involves the modification of an external software that cooperates with the OS to compute the best path to the destination which, in our case, varies according to the VPN_ID.

From this perspective, there are two models available in the literature. In the *piggyback* model a single routing daemon is able to compute the best path for all the VPNs. The routing daemon must be heavily modified since it must exchange, in its routing message with the peer routers, the VPN_ID of each destination. Vice versa, in the *virtual router* model, each router keeps several routing daemons active on the same machine. These daemons are completely independent (*ships in the night* approach) and each routing daemon exchanges only routing informations related to its VPN with the other peers.

The first model is probably more efficient, but it requires non-trivial modifications in the routing protocols. Vice versa, the second model allows the deployment of off-the-shelf daemons, with minimal modifications (you must assure that each routing daemon receives only the messages related to it). Our prototype uses a modified version of the Zebra [8] daemon that has a new starting parameter (the VPN_ID), which is used only to interact with the operating system (update routes or query for information). The remaining part of the daemon (routing messages, etc.) are kept unchanged. This modification allows the routing daemon to update on the part of the routing table that is related to its VPN, while to interact to unmodified daemons.

Since the network will have several routing messages flowing on it, each routing daemon bounds only to the virtual interfaces that are marked as belonging to the selected VPN: in other words, each tunnel carries only the routing messages that are related to the VPN it belongs to. Each router could have up to $N_{VPN} + 1$ routing daemons on it: the standard one (bound to all its physical interfaces) for the base network, and one addictional daemon for each locally served VPN.

IV. Related work

Several Internet Service Providers already have VPN provisioning in place; the most important router vendors have their solutions, and also the IETF community is working on that, trying to standardize a general solution. However, all the solutions available nowadays are based on the paradigm ”VPNs at the edge, traditional IP routing in the backbone”. Also solutions based on MPLS [9] can be seen as belonging to this paradigm, since VPNs are supported by creating a new set of label switched paths between ingress and egress routers so that multiple VPNs never share the same path.

Although non-existing in the marketplace, the idea of changing the router forwarding path in order to support VPNs natively has been examined by several projects in the literature. Among the others, the most important one is the Virtual Network Service project (VNS, [10]).

Although the technical solutions adopted in VNS seems to be quite similar to ours, the purpose of VNS is different. VNS was born to provide a private, secure, quality of service guaranteed channel between two end points. To do that, it uses IPSec (tunnel mode) in order to encapsulate the original IP packet. It follows that the intermediate routers on the path do not know the real (and private) address of the packet. Since the original IP address is hidden, the backbone routers should not need to know the VPN_ID of the packet. However, VNS provides Quality of Service guarantees on a per-VPN basis; therefore intermediate routers must know the VPN_ID of the packet. VNS, for instance, inserts the VPN_ID into an optional field of the outer IP header of the encrypted packet. Moreover, VNS modifies the routing protocols in order to support different paths (from the same couple ingress-egress routers) according to the VPN_ID. Therefore, the forwarding path of each router has to be modi-

fied in order to take into account both the destination address of the IPSec tunnel and the VPN_ID of the packet. It follows that VNS implements both a modified forwarding path (through multiple routing tables addressed by the VPN_ID) in order to support per-VPN routing and a mechanism to specify the VPN of each IP packet. Therefore VNS could support overlapped address spaces as well as we do, although this was not an objective of the project.

From the association between packets and VPNs, our present implementation uses a statically defined mapping between interfaces (also virtual, like tunnels) and VPNs, but this can be changed to a more sophisticated method (the one in VNS, or the MPLS tag, or even other ways) without changing the mechanisms that are used to forward IP traffic[2]. In other words, VNS wants to provide a way to create end-to-end VPN services; our project focuses particularly on the forwarding path and it wants to demonstrate an alternative way to create a VPN-aware IP network.

V. Conclusions

This paper presented the design and the implementation of an advanced support for provisioned virtual private networks, based on the FreeBSD operating system. The introduced new features allow FreeBSD to be adopted as an open-source mean for developing concurrent VPNs. In our implementation we modified the kernel functionalities (e.g. `socreate()`, `rtrequest()`, `ip_forward()`), the system calls providing an interface toward the kernel (e.g. `ioctl()`, `setsockopt()`, `sysctl()`), and many user-space applications (e.g. `ifconfig`, `route`, `netstat`, `zebra`).

The status of the current implementation is complete, and test-bed was ran at Politecnico di Torino, demonstrating that the implementation was working fine. Performances are not reported in this paper because there are absolutely no differences prior and after our modifications. In fact, FreeBSD already has support for multiple routing tables because it can handle several network-level protocols (IP, IPX, etc.) at the same time. The overhead of our VPN support can be seen like another network protocol, which results in a longer `switch` instruction into the forwarding path. The results confirms that the same operating system forwards the same number of packets with or without our modifications.

However some issues need to be further investigated. The main problem concerns the ARP module of standard FreeBSD. Since ARP entries are cached within the IP base routing table, this creates a conflict when a VPN access router is connected to multiple sites that are using the same address space. The virtualization of ARP caches and ARP lookups is needed (similarly to the virtualization of tables and table lookups, as described in Section III) in order to make the implementation more flexible. The second issues relates to the standard GRE module. By now, it is not possible to define more than one tunnel bounded to the same couple of physical interfaces between two peer VPN routers. Obviously, this hampers the applicability of our implementation, since it is not possible to deploy two parallel tunnels for two distinct VPNs without using different physical (i.e. outer) addresses. Such problem could be overcome by patching the GRE support, to integrate the adoption of the GRE key field. By mapping the key field to the `gif` VPN identifier, the parallel tunnels would be still distinguishable.

Besides this issues, further work can be undertaken in many area. For instance, the Zebra support for multiple routing table could be improved. Our current implementation requires the instantiation of a `zebra` router manager daemon (and a corresponding `ospfd` routing daemon) for each defined table. It would be more manageable to have a single router manager and let the routing daemons to specifies the table of interest for the routing updates. This requires a deeper modification (with respect of the current status), since the communication protocol between the daemons and the manager should be extended.

A second major improvement concerns the identification of VPN traffic at the access side. Currently, all packets incoming from a tagged physical interface are associated to a single VPN. In case of a client site belonging to multiple VPN, the site access router must be connected to multiple interfaces of the provider access router. Further, the site access router must be able to distribute client packets of different VPNs towards the different interfaces it is attached to. This requires additional capabilities from the site access router, hence hampering the transparency for the client. To this aim, a more sophisticated approach could be used for traffic identification. Multiple traffic filters could be applied to the access interface (instead of a single tag, which is logically equivalent to a single wild-card filter). Filters could be used to identify the membership of a packet to a given VPN on the basis of protocol fields of the TCP/IP headers. This would also allow a greater granularity to the traffic identification operation. This solution would allow the site access router to be attached to a single interface toward the backbone network, and would simplify its task: it should have a single default route for all non local traffic (rather than distributing packets on several outgoing interfaces).

Finally, other possible evolutions are the support of IPSec tunnels to allow secure VPNs when needed and the integration of a QoS module (e.g. ALTQ [11]) with the virtual forwarder, to allow QoS-based forwarding on per-VPN basis.

[2]In this case the current implementation of the virtual routing daemon must be changed as well because it relies on different interfaces (i.e. tunnels) to distinguish the routing messages belonging to different VPNs.

Acknowledgments

The authors would like to thank Angelo Calafato for his great job in the implementation of the prototype.

References

[1] B. Gleeson, et al., *A Framework for IP Based Virtual Private Networks*, IETF RFC 2764, Feb. 2000
[2] R. Callon (ed.), et al., *A Framework for Layer 3 Provider Provisioned Virtual Private Networks*, IETF Internet Draft, Apr. 2002
[3] D. Farinacci, et al., *Generic Routing Encapsulation (GRE)*, IETF RFC 2748, Mar. 2000
[4] G. R. Wright, W. R. Stevens, *TCP/IP Illustrated, vol 2: The Implementation*, Addison-Wesley, 1995
[5] S. J. Leller, et al., *The Design and the Implementation of the 4.3BSD UNIX Operating System* Addison-Wesley, 1989
[6] A. Calafato, *Architectural Choices for Developing Virtual Networks* (in Italian), Master thesis, Politecnico di Torino, Jan. 2002
[7] FreeBSD 4.4 patches, On-line at http://softeng.polito.it/freebsd/
[8] Zebra Project Page, On-line at http://www.zebra.org
[9] E. Rosen, et al., *BGP/MPLS VPNs*, IETF RFC 2547, Mar. 1999

[10] L.K. Lim, et al., Customizable Virtual Private Network Service with QoS, Computer Networks, vol. 36, no. 2-3., pp. 137-151, Jul. 2001
[11] K. Cho, *A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX based Routers*, Usenix 1998, New Orleans, Louisiana, USA