Universidad de Mendoza

Facultad de Ingeniería

Tesis de Maestría en Teleinformática

Red Privada Virtual sobre Mensajería Instantánea

Ing. Juan José Ciarlante

Directores de Tesis:

Magíster en Ing. Electrónica Hugo Etchegoyen

Ing. Osvaldo Rosso

Mendoza, Noviembre de 2005

Instituto de Informática, Facultad de Ingeniería, Universidad de Mendoza. Peatonal Emilio Descotte 750, Mendoza, Argentina.

A quienes defienden la libertad del espíritu, mente, cuerpo y *bits* ...

Agradecimientos

Quiero agradecer especialmente a Clau por su invaluable ayuda como científica en la elaboración y corrección de este documento; nuevamente a ella, Emi y Pablito por su amor y apoyo incondicionales.

A mis colegas de Universidad de Mendoza, especialmente Diego por su paciencia y ayuda como amigo y compañero. A Alfredo y Salvador por apoyarme siempre en todos mis proyectos.

Y finalmente a la comunidad de desarrollo de software libre, gracias a la cual he podido aprender y dar más de lo que jamás hubiera imaginado ...

Índice de contenido

1. Resumen

El montaje de un *VPN* (*Virtual Private Network*: Red Privada Virtual) típicamente requiere de la disponibilidad de **al menos un punto de presencia "expuesto" a la red pública**, relativamente estático en cuanto a su direccionamiento y/o su *FQDN* (*Full Qualified Domain Name*: nombre del nodo incluyendo dominio) hacia el cual el *initiator* (lado iniciador de la negociación) comienza la gestión del vínculo *VPN*.

El presente trabajo demuestra que lo anterior no es condición necesaria si se dispone de las herramientas adecuadas:

- Un gestor de *VPN* con único flujo de gestión+tráfico tal como **OpenVPN[1]**
- Un flujo bidireccional de baja latencia y buen ancho de banda entre dos puntos: un sistema de *I.M*. (*Instant Messaging*: mensajería instantánea) tal como **Jabber[2]**.

En efecto: se demuestra, con resultados que incluyen mediciones de latencia y *bandwidth* (ancho de banda efectivo) que es posible establecer una **Red Privada Virtual sobre Mensajería Instantánea** totalmente operativa utilizando un *gateway* (entidad intermediaria) desarrollado a tal efecto.
En este escenario el "servidor" de *I.M.* (público, externo a la organización) hace las veces de *SWITCH* retransmitiendo los mensajes *I.M.* entre los extremos cliente, estos mensajes contendrán los datagramas del flujo *VPN* codificados como si se tratara de una conversación entre dichos extremos cliente.

Como complemento a los cambios hechos al núcleo de OpenVPN en el manejo de *sockets* (puntos de conexión usados en la programación de

aplicaciones de red) surgió el soporte de **OpenVPN para IPv6** (*Internet Protocol* versión 6) y la **optimización TCP-sobre-TCP.**
Ambos cambios **ingresaron al repositorio de desarrollo oficial del proyecto OpenVPN en Julio de 2005.**

Como valor agregado final, dado el contexto operativo cada vez más hostil para la "privacidad en la red" de nuestro país[3] y el mundo, resulta sumamente interesante poder utilizar el *gateway* desarrollado para montar un *VPN stealth* (difícilmente detectable) **sin ser visible la dirección IP pública del otro extremo**.

NOTA: la presente tesis se completa con dos archivos adicionales con el código fuente completo, el cual no se incluye en la presente por razones de formato y de tamaño. Ver 5.1.

---

2. Introducción

2.1. Objetivos

- Mostrar que es posible **vulnerar el perímetro de seguridad de una red mediante el establecimiento de un *VPN* potencialmente hostil** cuando está permitido el uso de **servicios de mensajería instantánea externos.**
- Mostrar que el *VPN* establecido tiene propiedades de **"invisibilidad" de la dirección IP del extremo remoto.**
- Contribuir a la comunidad de Software Libre con el presente desarrollo, herramientas y resultados.

Además, como objetivo secundario de índole personal, se propone **demostrar que es posible realizar una tesis de posgrado utilizando 100% software libre en todas sus fases: herramientas técnicas, diagramación, documentación, etc.**[1]

Cabe notar que una de las motivaciones originales fue encapsulamiento de IP en XML propuesto en la RFC3252: Binary Lexical Octet Ad-hoc Transport [4][2].

2.2. Marco Teórico

2.2.1. *VPN*: *Virtual Private Network*

Un *VPN* se puede definir como un **vínculo virtual protegido** entre dos

1 Se podría hablar del grado de libertad obtenido como consecuencia de este enfoque, pero no es el objeto del presente.

2 Notar la fecha de publicación: ***April 1st***, 2002.

redes.

Dicho vínculo tiene las siguientes propiedades:

- ***V***irtual
  Porque no existe un vínculo de físico ni de enlace (capa1 y capa2 del modelo OSI) entre los extremos, sin embargo se **comporta como si existiera un vínculo punto a punto**.

- ***P***rivate (privado) en realidad debería ser: *protected* (protegido)
  Porque provee servicios de seguridad de manera que el **vínculo protegido posee un nivel de seguridad mucho mayor que el de la red usada como transporte**.

  Los servicios de seguridad provistos como mínimo deben ser:

  - Autenticación
  - Control de acceso
  - Confidencialidad
  - Integridad

- ***N***etwork (red)
  Porque el servicio trabaja en capa de red (capa3 del modelo OSI) a diferencia, por ejemplo, de *https* (http seguro) que trabaja en sesión~aplicación (capa de 5~7 del modelo OSI) ó IEEE 802.11i que lo hace a nivel de enlace (capa2 del modelo OSI).

Las *VPN* se han convertido en la *estrella* de la conectividad en los últimos tiempos; presentan una opción muy válida para reemplazar enlaces *WAN* (Wide Area Network: red de área amplia) que brindan conectividad dentro de la organización.

Asimismo permiten el despliegue de nuevos escenarios, tales como el acceso intermitente a la red interna de la organización desde puntos de conectividad no fijos de la red pública[1].

Sus ventajas más notables son flexibilidad, escalabilidad, bajo costo de equipamiento y razonablemente alto nivel de seguridad.

Su principal desventaja es el mayor grado de complejidad de montaje dadas las exigencias de seguridad a la que están sometidas por tener que brindar altos niveles de acceso a la red interna estando expuestas a la red pública.

2.2.2. VPN: Arquitecturas e implementaciones

En la Tabla 1 se comparan algunas implementaciones y sus características: estándard de *IETF* (*Internet Engineering Task Force*[2]), plataformas (Linux/Un*x, WinXX y otros dispositivos dedicados), nivel de interoperabilidad, gestión y aplicación de *policy* (políticas de acceso), servicios de seguridad (soporte de PSK[3], PKI[4], algoritmos criptográficos (3DES, AES, MD5, SHA1, SHA2, etc), capacidad de atravesar routeadores que implementan *NAT (Network Address Translation*: traducción de direcciones de red para su visibilidad pública), protocolo de gestión, protocolo de encapsulamiento de tráfico de datos, capacidad de gestión de parámetros de red y/o de aplicación (autoconfiguración de direcciones IP, DNS, etc).

1 Este modo de acceso se suele denominar "*road warrior"*

2 Organización abierta, internacional encargada de generar documentación relevante al diseño, operación, gestión y evolución de la Internet global y sus procotolos. Los documentos incluyen estándares (RFCs), *best practices* (BPs - consejos prácticos) y otros.

3 PSK: Previously Shared Key

4 PKI: Public Key Infrastructure: autenticación RSA/X509 con Cert. Authority, etc

Impl.	Estándar	Plat.	Inter-op.	Pol. Acc.	Servicios seguridad	NAT ok	Proto. Gestión	Proto. DATA	Gestión Param. de red
IPSec[5]	Sí RFC2401, RFC2408, RFC2409	Un*x/ Win2k+ / otros	Muy Alta	Sí	PSK, PKI, varios ALGs,	Difícil	*IKE udp 500*	*ESP/ AH*	No[6]
OpenVPN	No	Un*x/ Win2k+	N/A	No	PSK, PKI, varios ALGs	Sí	TLS/udp 1194 (configurable)		Si
PPTP	Sí[7] RFC2637	Un*x/ Win*/ otros	Media	No	Muy pobres (sólo RC4) userpass-> *session key*	No	*tcp 1720*	*GRE*	Sí (PPP)

Tabla 1: Implementaciones de VPN y sus características

El establecimiento y uso de un vínculo *VPN* típicamente requiere de dos flujos distintos:

1. **Gestión de la sesión:** autenticación de las partes (a través de un canal asegurado) y parámetros/atributos de los vínculos protegidos.

   Esto se ve claramente implementado en el conjunto de protocolos *IPSec* [6] por el protocolo *IKE* (*Internet Keying Exchange*: Protocolo

5 Coloquialmente llamado "IPSec", IPSec es en realidad un conjunto de protocolos compuesto por lo protocolos de bajo nivel: ESP y AH (estos serían estrictamente IPSec); y otros como IKE para gestión de asociación, intercambio de claves, etc.

6 Usando L2TP/IPsec es posible contar con una gestión más amplia de parámetros

7 Lamentablemente la implementación de Microsoff utiliza extensiones propietarias con un mediocre diseño de servicios de seguridad. Se han analizado y documentado mecanismos para quebrar la encriptación provista por MS/PPTP [5] en corto tiempo, sin embargo esto no ha evitado su uso masivo.

para intercambio de claves) [7] [8].
Como resultado de la gestión de *IKE* se crean claves de sesión "efímeras" que servirán como "semilla" (*key material*) para derivar las claves requeridas por los algoritmos de cifrado /autenticación que protegerán el tráfico de datos (por ej. AES 128bits + SHA2 256bits).

1. **Tráfico de datos:** mensajes de datos protegidos por los servicios de seguridad provistos, por ejemplo *ESP* (*Encapsulated Security Payload*: carga de seguridad encapsulada)

Debido a que la mayoría de las implementaciones de *VPN* usan flujos **distintos** (en cuanto a protocolo, puerto, etc.) para gestión de sesión y tráfico de datos, resulta sumamente **complicado** (sino imposible) **encaminar ambos a través de dispositivos que realicen *NAT*,** situación cada vez más común a medida que la escasez de direcciones IP "públicas" se agrava[9].

El *IETF IPSec WG* (grupo de trabajo del estándar IPSec) ha elaborado recientemente la RFC3947 [10] que describe un mecanismo que encapsula los paquetes *ESP* en mensajes *IKE* para así salvar el problema de *NAT* para *IPSec* (denominado *IPSec NAT-Traversal*). Sin embargo el *IPSec responder*[1] debe estar atendiendo los puertos UDP 500/4500 por lo tanto no es posible tener varias "instancias" distintas de *IPSec* en dicho terminador, lo cual imposibilita hacer *NAPT (Network Address and Port Translation*[2]) desde la dirección pública hacia direcciones privadas basado en el puerto destino del primer mensaje enviado por el *IPSec*

1 Equipo que **recibe** el primer mensaje de establecimiento de VPN

2 Similar a *NAT* pero a nivel de transporte, lo cual involucra conocer semánticamente el protocolo para realizar el cambio de puertos, mantener el estado de cada asociación o conexión, etc.

initiator.

En la Figura 1 se observan los componentes de *IPSec* en forma de mapa conceptual[1]; se puede apreciar el importante grado de complejidad que contemplan las especificaciones del conjunto de protocolos *IPSec*; esta complejidad efectivamente afecta el despliegue de *IPSec* como implementación de *VPN.*

1 Se eligió mostrar la arquitectura de IPSec, a pesar de no haberse elegido como base del trabajo, debido a que representa muy bien los componentes requeridos para una implementación correcta de VPN.

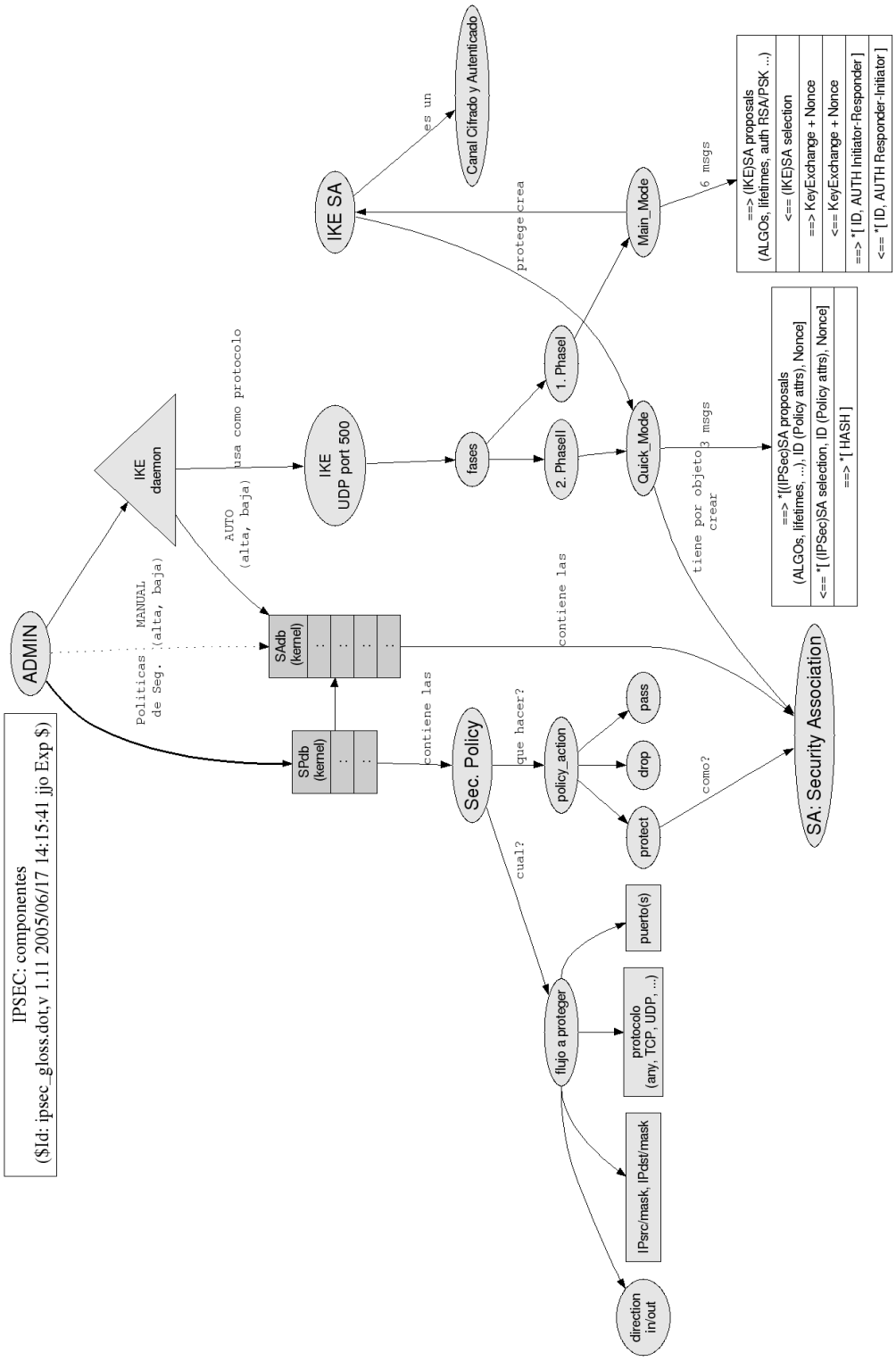

Figura 1: Componentes de IPSec: Relación funcional

En el presente trabajo se optó por usar **OpenVPN** principalmente debido a las siguientes características:

- **Único flujo para encaminar flujos de control+tráfico**

  Debido a que los mensajes de control y transporte de datos utilizan el **mismo flujo de paquetes**[1] es capaz de atravesar *NAT* trivialmente en cualquier sentido.

- **Software libre [11]**

  El desarrollo de la tesis modifica el código fuente de OpenVPN para optimizarlo para el uso propuesto, por lo que la disponibilidad del código fuente y la capacidad de analizarlo, estudiarlo y modificarlo para cualquier fin son propiedades necesarias.

- **En activo desarrollo**

  Siendo que uno de los objetivos es contribuir a la comunidad de software libre, es crítico elegir un proyecto que se encuentre en activo desarrollo para que el esfuerzo pueda ser aprovechado por un conjunto más amplio de usuarios y desarrolladores.

- **Implementación simple, uso simple**

  OpenVPN tiene un excelente balance entre seguridad, simpleza y funcionalidad.

- **Disponibilidad para diversas plataformas (Linux, xBSD y algunas otras)**

1 Por defecto utiliza paquetes UDP, puerto 1194.

2.2.3. *VPN*: Topología

El montaje de un *VPN* típicamente requiere la instalación de un *endpoint* (punto final de "conexión") en las periferia de seguridad de las redes que se pretenden vincular.

Para esto es necesario disponer de un *flujo bidireccional de baja latencia* entre los endpoints internos. No hace falta buscar demasiado para encontrar con un excelente "proveedor" del mismo: el servicio de **mensajería instantánea (*I.M.*)**.

2.2.4. *I.M. (Instant Messaging)*: un canal bidireccional de baja latencia

La mensajería instantánea se ha convertido en la *killer application* (aplicación muy exitosa) de los últimos tiempos, especialmente entre los más jóvenes [12].

En la mayoría de las implementaciones actuales de mensajería instantánea la posibilidad de transportar una carga (*payload*) distinta al texto de la conversación depende del soporte específico del protocolo correspondiente.

El presente trabajo codifica los paquetes de transporte del *VPN* usando *base64* (encapsulamiento de mensajes de contenido arbitrario en texto ASCII) y encapsula los mensajes de texto resultantes como si se tratara de una conversación convencional.

También sería posible utilizar *esteganografía*[1] para producir un flujo de conversación menos "sospechoso" al análisis de tráfico, por ejemplo se podría construir un diccionario de palabras "comunes" asociadas

1 Rama de la criptografía que trata sobre la ocultación de mensajes para evitar que se perciba la existencia del mismo.

bidireccionalmente con bytes ó conjunto de bytes[1] en forma bidireccional y unívoca.

2.2.5. I.M.: Jabber y *XMPP*

La red Jabber utiliza *XMPP* (*eXtensible Messaging and Presence Protocol*), el cual es un protocolo abierto para mensajería instantánea basado en *XML* (*eXtended Markup Language*). El transporte por defecto es TCP, puerto 5222.

***XMPP* ha sido estandarizado por IETF en las RFC3920 [13] y RFC3921 [14][2].**

XMPP tiene las siguientes características importantes[2]:

- **JID** (Jabber ID): es la dirección del *endpoint* de la red Jabber

  Su sintaxis es: ***usuario@host/recurso***.

  JID puede considerarse como la "instancia de conexión" de un **mismo usuario,** factible de diferenciarse arbitrariamente cambiando ***recurso*** , por ejemplo:

  luser@jabber.org/Portatil
  luser@jabber.org/Oficina
  luser@jabber.org/Celular

  Esto permite **identificar unívocamente** dichos *endpoints* para intercambiar mensajes, utilizando una sola "cuenta de usuario".

- Red descentralizada de servidores

1 El resultado de una captura de tráfico requeriría de un analizador gramatical para discernir una charla entre humanos de un flujo de palabras pseudo-caóticas.

2 *XMPP* es el único protocolo de *I.M.* que ha sido estandarizado por IETF, no existiendo al momento ningún proceso de estandarización de otros.

Jabber no posee un “servidor central” sino que utiliza una “red de servidores”, cada uno de los cuales contiene sus cuentas de usuario. El protocolo contempla tanto la comunicación *c2s* (*client-to-server*) como la *s2s* (*server-to-server*).

Cada JID que ingresa a la red se conecta y autentica con su servidor[1] creando el canal a través del cual se intercambiarán **todos** los mensajes que envíe o reciba dicho JID (comunicación *c2s*).

El servidor Jabber intercambiará mensajes entre JIDs de sus **propios** usuarios sin otros intermediarios. Los mensajes intercambiados con JIDs de **otros dominios** serán enviados/recibidos a través del servidor correspondiente, el cual se “descubre” consultando al *DNS* (servicio de nombres) por el registro *MX* (*Mail Exchanger*: nodo intercambiador de correo) del dominio correspondiente.

- Soporte de tráfico protegido (cifrado, integridad, etc.)

  TLS/SSL (*Transport Layer Security Protocol*: protocolo de seguridad para capa de transporte) para proteger el canal cliente-servidor frente a ataques tales como *sniffing* (captura de tráfico), *man-in-the-middle* (intercepción o alteración de tráfico, robo de identidad, etc.).

- Extensible por diseño

  Mediante un proceso de discusión abierto (similar a las *RFC*s de IETF) se gestionan las mejoras al protocolo: *JEPs* (*Jabber Enhancement Proposals*: propuestas de mejoras a Jabber) [15]

Puede mencionarse que recientemente ***Google*** puso en producción su sistema de mensajería instantánea (talk.google.com) el cual **usa XMPP como protocolo de *I.M.***.

1 El servidor se obtiene del registro *MX* del *DNS* correspondiente al dominio de la cuenta.

2.2.5.1. XMPP: Tipos de mensajes

XMPP provee varios tipos de mensajes (llamados “elementos *XML*” por en la documentación de Jabber) para la comunicación *c2s*:

Elemento	***Tag[1] de XML***	***Tipos***
Mensaje	`<message>`	normal, chat, groupchat, headline, error
Consultas de infomación	`<iq>`	get, set, result, error
Presencia	`<presence>`	available, unavailable, subscribe, subscribed, unsubscribe, unsubscribed, error

2.2.5.2. Jabber: Ejemplo de conversación

A continuación se detallan los mensajes intercambiados en una conversación hipotética entre **A**licia y **B**runo.

Bruno es notificado que Alicia está disponible:

```
<presence from='alicia@greenpeace.org/Trabajo'
    to='bruno@savetheplanet.org/Home'>
  <status>Online</status>
  <priority>2</priority>
</presence>
```

Bruno envía un mensaje rápido a Alicia:

```
<message type='chat' from='bruno@savetheplanet.org/Home'
    to='alicia@greenpeace.org'>
  <thread>01</thread>
  <body>Hola Alicia! ... viste las noticias de la tala indiscriminada en
nuestra selva ? :-( </body>
</message>
```

1 El *Tag* de *XML* define el tipo de dato ó elemento dentro de la estructura del documento.

Alicia responde:

```
<message type='chat' to='bruno@savetheplanet.org/Home'
    from='alicia@greenpeace.org/Trabajo'>
  <thread>01</thread>
  <body>No lo puedo creer ...</body>
</message>
```

Bruno envía a Alicia el *URL* (*Uniform Resource Location*: comúnmente llamado “dirección en Internet”) de la noticia:

```
<message type='chat' from='bruno@savetheplanet.org/Home'
    to='alicia@greenpeace.org/Trabajo'>
  <thread>01</thread>
  <body>Fijate la noticia en ...</body>
  <x xmlns='jabber:x:oob'>
    <url>http://www.we-dont-lie.com/news/argentina.html</url>
    <desc>Argentinian forest under danger</desc>
  </x>
</message>
```

2.2.5.3. Jabber: un “router XML”

En efecto **la red Jabber conforma una inter-red** en la cual se pueden distinguir sus componentes: *address* (dirección/identificación), paquetes, *I.S.* (*intermediate systems*: sistemas intermediarios), *E.S.* (*end systems*: sistemas finales/terminales), *forwarding* (reenvío).

El la Tabla 2 se muestra una analogía entre los elementos de una red IP y la red Jabber.

componente	*red IP*	*red Jabber*
address	**dirección IP**	**Jabber ID (JID)**: usuario@dominio/recurso
paquetes	datagramas IP	mensajes *XML*
I.S.	*routers* (encaminadores)	Jabber *servers*
E.S.	*hosts* (sistemas finales)	Jabber *clients*
forwarding	*next* HOP en base a dirección IP destino	Jabber *server* en base a DNS MX del dominio del JID destino

Tabla 2. Analogía entre una red IP y la red Jabber

2.2.6. *VPNs* y Mensajería Instantánea

La propuesta técnica del trabajo consiste en **encapsular los paquetes generados por OpenVPN en mensajes XML** que se **intercambiarán usando la red Jabber (*XMPP*).**

3. Desarrollo

3.1. Escenario de uso

En la Figura 2 se muestra el escenario típico de uso de OpenVPN: el flujo UDP/puerto1194 se usa para establecer la asociación entre los *security gateways* (intercambiadores seguros) OV1 y OV2 para proteger el tráfico entre las redes Red1 y Red2. R1 y R2 son los *routers* de borde de sendas redes.

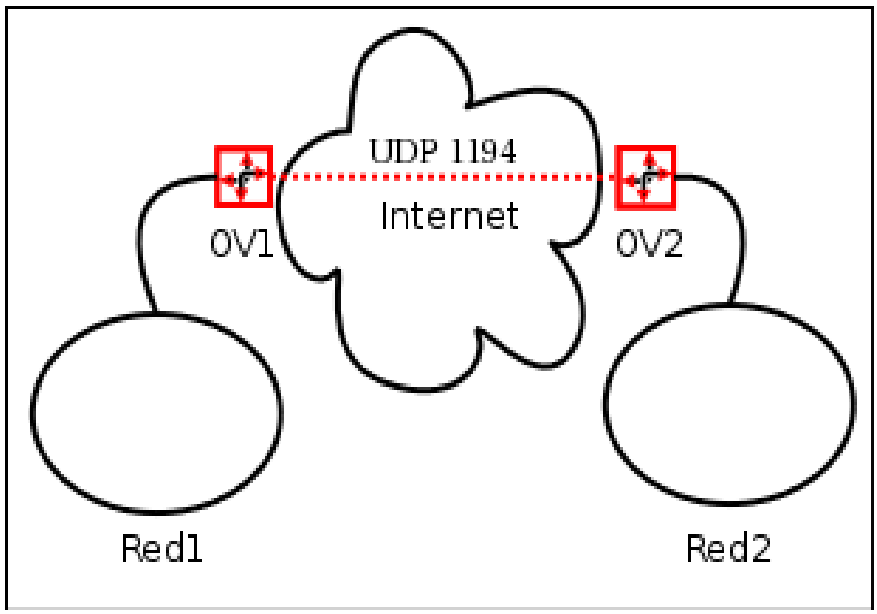

Figura 2. OpenVPN sobre UDP (nativo)

En la Figura 3 se muestra un diagrama con la nueva topología resultante: los extremos OV1 y OV2 se hallan dentro del perímetro de sendas redes intercambiando mensajes *XMPP* a través del servidor Jabber en vez de hacerlo directamente como en la Figura 2.

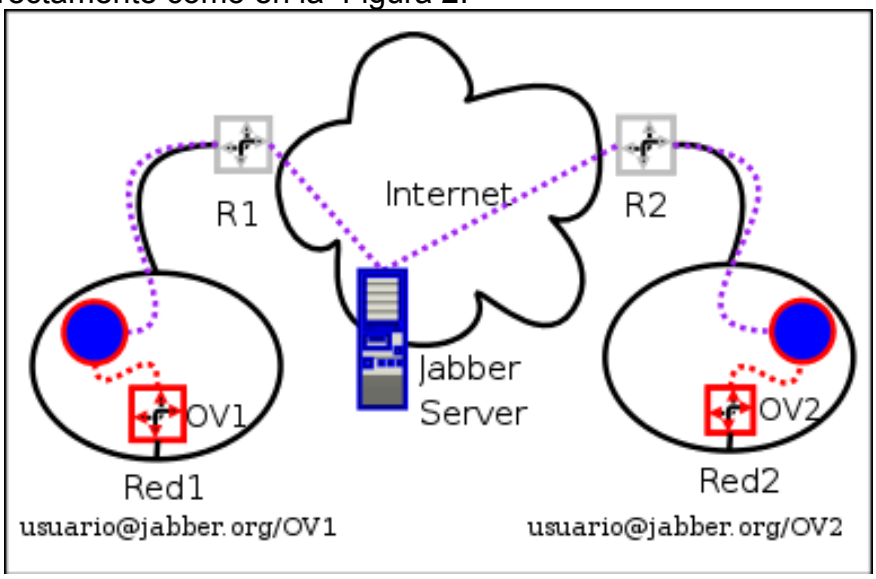

Figura 3. OpenVPN sobre XMPP (OXG)

Los componentes nuevos más importantes de la Figura 3 son los *gateways* OpenVPN/XMPP (círculo rojos con relleno azul) denominados *OXG* (OpenVPN XMPP Gateway).

OXG realiza la codificación/decodificación de los paquetes UDP en mensajes *XMPP;* éste inicia previamente una sesión autenticada en el servidor Jabber como `usuario@jabber.org/OV1` y `usuario@jabber.org/OV2`.

Nótese que si se realizara una captura de tráfico en R1 y R2 se observaría un flujo de paquetes *I.M.*, típicamente considerado “inofensivo” e incluso “productivo” por las políticas de seguridad de buena parte de las organizaciones[16].

3.2. Encapsulamiento

En la Figura 4 se muestran las cabeceras nativas de los protocolos correspondientes cuando se usa OpenVPN en forma nativa sobre UDP y en la Figura 5 el resultante luego del encapsulamento en *XMPP*.

Figura 4: Encapsulamiento OpenVPN/UDP (nativo)

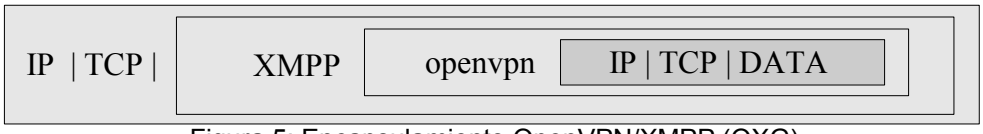

Figura 5: Encapsulamiento OpenVPN/XMPP (OXG)

3.3. Desarrollo original: *OXG - Gateway UDP <---> XMPP*

OXG (*gateway OpenVPN-XMPP)* es el componente central de conectividad desarrollado en el presente trabajo.

3.3.1. Operación de gateway

OXG cumple el rol de *gateway* entre OpenVPN (protocolo *UDP*) y Jabber (protocolo *XMPP*), encapsulando los paquetes UDP desde OpenVPN en mensajes XMPP y viceversa tal como se puede ver en las Figura 5 y Figura 6.

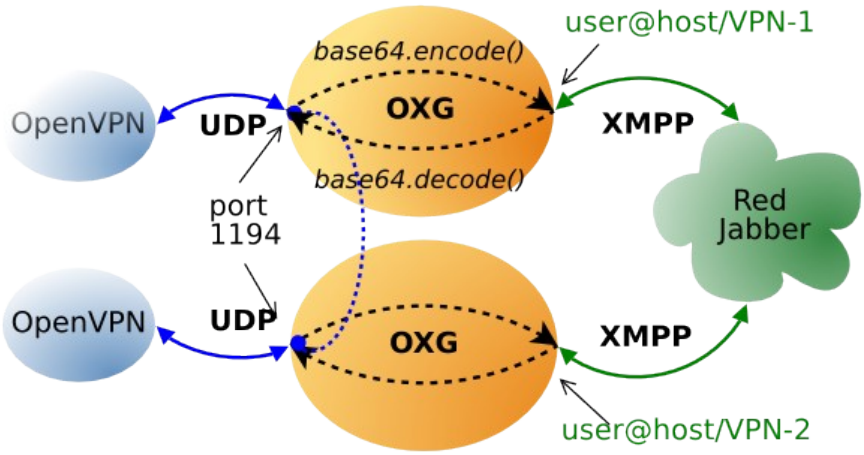

Figura 6: OXG - OpenVPN XMPP Gateway

OpenVPN percibe estar “conectado” al otro como si efectivamente estuviera en el puerto UDP donde está el gateway OXG (representado con la línea punteada central en azul)

El núcleo del *OXG* se encarga de:

- Codificar los paquetes UDP recibidos desde OpenVPN usando *base64* y con el resultado construir y enviar un mensaje XMPP al JID destino.
- Decodificar los mensajes XMPP recibidos desde el JID remoto, extraer su *body* (cuerpo del mensaje), decodificar su contenido *base64* y enviar el resultado vía UDP a OpenVPN.

Cabe notar que el proceso de codificación solamente transforma los paquetes OpenVPN convirtiéndolos a ASCII para usarlos como cuerpo de los mensajes *XMPP* sin realizar ninguna operación de cifrado, autenticación, etc. por ser innecesaria dado que dichos **servicios de seguridad YA están provistos por OpenVPN *per se*.**

Desde un punto de vista más generalizado OXG realiza una *traducción de direcciones (similar a NAT) entre dos redes distintas:* la red IP/UDP y la red Jabber.

3.3.2. Entorno de desarrollo: Python

Se eligió desarrollarlo usando Python [17] por las siguientes características del lenguaje y el entorno de programación:

- *OOP* (*Object Oriented Programming*: Programación Orientada a Objeto)

  Además de las características propias de *OOP*, el manejo de excepciones es especialmente útil para desarrollar aplicaciones que utilizan servicios de red ya que la presencia de "fallas" **es** una condición de diseño.

- Cortos tiempos de desarrollo[1]

- Excelente soporte de bibliotecas para servicios de red

  La mayoría de protocolos de red utilizados se pueden encontrar implementados en la biblioteca estándar de Python o en otras adicionales disponibles libremente.

3.3.3. *OXG*: Estructura de la aplicación

En la Figura 7 se muestra las clases principales que conforman *OXG*[2].

1 El mismo prototipo inicial desarrollado en Python en media hora había requerido unas 6 horas hacerlo usando C/POSIX

2 Se optó por usar nomenclatura "internacional" (inglés) para los nombres de clases, atributos, métodos, etc. para lograr una mayor difusión del desarrollo una vez publicado.

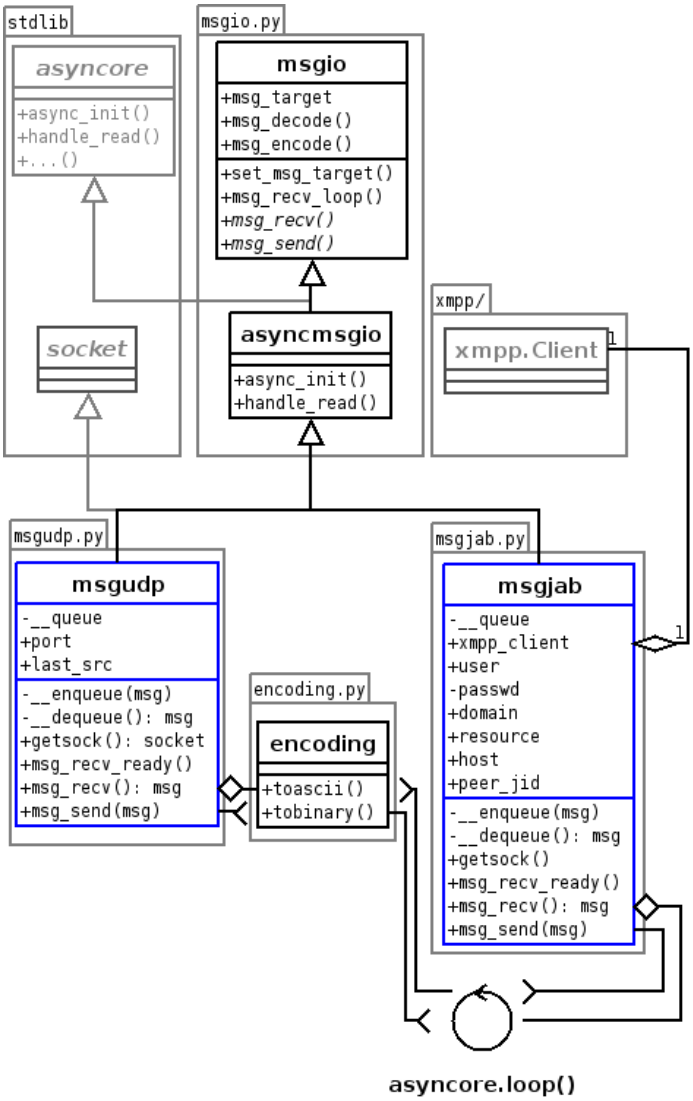

Figura 7: OXG - Diag. UML aproximado

Se eligió la biblioteca XMPPpy [18] como implementación de *XMPP* para Python.

A continuación se describe el rol de los componentes principales (ver Figura 7):

- *class* `msgio [msgio.py]`

  Generalización de las clases específicas para intercambio de mensajes (UDP, XMPP), su interfaz contiene métodos tales como `msg_send()`, `msg_recv()`, etc.

  Los métodos más importantes son:

  - `msg_send()` ***virtual***

    Método virtual para enviar un mensaje al *endpoint* remoto. Debe ser implementado por la clase derivada dependiendo del transporte -- UDP ó XMPP-- que se trate.

  - `msg_recv()` ***virtual***

    Método virtual para recibir un mensaje desde el *endpoint* remoto. Debe ser implementado por la clase derivada dependiendo del transporte --UDP ó XMPP-- que se trate.

  - `msg_recv_ready()` ***virtual***

    Método virtual a ser invocado por `handle_read()` cada vez que el descriptor de conexión esté listo para leer (típica implementación de bucle `select()/poll()`).

    Debe ser implementado por las clases derivadas.

  - `msg_recv_loop()`

Invocada por las clases derivadas luego de la recepción de uno o varios mensajes, extrae cada mensaje llamando a `self.msg_recv()` y lo envía a `self.msg_target` realizando la decodificación/codificación previa si están asignados los métodos `self.msg_decode() / self.msg_target.msg_encode()`.

Un extracto de código se muestra a continuación (chequeo de error, etc. omitidos) :

```
while True:
 m=self.msg_recv()
 if (m==None):
 break
 try:
 if (self.msg_decode): m=self.msg_decode(m)
 if (self.msg_target.msg_encode): m=self.msg_target.msg_encode(m)
 self.msg_target.msg_send(m)
 except:
 continue
```

- `set_msg_target()`

  Asigna el destinatario de los mensajes recibidos y opcionalmente las funciones de codificación/decodificación.

  Usado en el cuerpo de `main()` de la siguiente manera:

```
udp=msgudp.msgudp()
jab=msgjab.msgjab(...)
udp.set_msg_target(jab, encoding.tobinary, encoding.toascii)
jab.set_msg_target(udp)
```

- *class* `asyncmsgio`, derivada de `msgio [msgio.py]`

  Agrega a `msgio` la interfaz necesaria para poder registrarse en `asyncore` (de la biblioteca estándar de python).

  El método más importante implementado es `handle_read()` invocado por el motor de asyncore cada vez que el *socket* (descriptor del punto

de acceso a los servicios de red) es legible sin espera:

```
def handle_read(self):
    self.msg_recv_ready()
    self.msg_recv_loop()
```

- *class* `msgjab`, derivada de `asyncmsgio [msgjab.py]`

  Implementa el intercambiador de mensajes *XMPP*, contiene un objeto `xmpp.Client` (de la biblioteca XMPPpy) encargado de realizar la interacción con el servidor Jabber (autenticación, intercambio de mensajes, etc).

  No se pudo implementar derivando `msgjab` de la clase `xmpp.Client` debido a la implementación asíncrona de la misma; de allí que `msgjab` contenga una instancia de `xmpp.Client` en la variable `xmpp_client`.

  - `__enqueue()`, `__dequeue()`:

    Inserta/extrae mensajes de `self.__queue[]`.

    Estas funciones de encolado son necesarias debido a que **no existe una relación uno-a-uno entre las lecturas realizadas con `read()` sobre el *socket* TCP desde el server Jabber y los mensajes *XMPP* extraíbles** (un `read()` puede "contener" varios mensajes *XMPP* o viceversa, varios `read()` pueden ser necesarios para completar un mensaje *XMPP*).

  - `getsock()`

    Invocada por el "motor" `asyncore`, retorna el objeto `socket.socket` correspondiente a la conexión con el servidor Jabber.

  - `msg_recv_ready()`

Invoca a `self.xmpp_client.Process()`, función principal de `xmpp.Client` para construir los mensajes *XMPP* recibidos como resultado de `read()` del socket de conexión al server Jabber.

- `msg_recv()`

  Invocado por el lazo `msgio::msg_recv_loop()`, desencola **de a uno** los mensajes *XMPP* acumulados por `msg_recv_ready()`.

- `msg_send()`

  Implementa el envío de mensajes XMPP, invoca `self.xmpp_client.send()` para hacerlo.

- *class* `msgudp`, derivada de *`asyncmsgio,socket [msgudp.py]`*

  Implementa el intercambiador de paquetes UDP.

  - `__enqueue()`, `__dequeue()`:

    Inserta/extrae mensajes de `self.__queue[]` .

  - `getsock()`

    Invocada por el "motor" `asyncore`, debe retorna el objeto `socket.socket` donde *OXG* espera los paquetes UDP enviados por el OpenVPN remoto.

  - `msg_recv_ready()`

    Extrae el contenido de datos del paquete UDP enviado por OpenVPN usando `recvfrom()` y lo encola para ser extraído posteriormente por `msg_recv()`.

Otra función importante que cumple este método es memorizar el endpoint UDP remoto en la propiedad `self.last_src`.

- `msg_recv()`

  Invocado por el lazo `msgio::msg_recv_loop()`, desencola el contenido de los paquetes UDP encolados previamente por `msg_recv_ready()`.

  Dicho contenido **es** el paquete OpenVPN que será codificado y encapsulado en un mensaje XMPP.

- `msg_send()`

  Implementa el envío de paquetes UDP hacia OpenVPN remoto, invoca `sendto(self.last_src)`

- *class* `encoding [encoding.py]`

  Encapsula los métodos `tobinary()` y `toascii()` utilizando las funciones provistas por la biblioteca estándar en el módulo `base64`.

- `main() [main.py]`

  Invoca a `parseconf.parseargs()` para leer el archivo de configuración pasado como `argv[1]`, cargando las propiedades del objeto global `conf`:

  - `conf.my_jid`

    El JID de este *Jabber endpoint*.

  - `conf.peer_jid`

    El JID del *Jabber endpoint* remoto.

- `conf.server` (opcional)

  Nombre/dirección IP del servidor Jabber, su valor por defecto es el host obtenido del JID:

  usuario**@host**/resource

- `conf.password`

  La clave usada para autenticar `conf.my_jid` frente a `conf.server`.

- `conf.udpport`

  Puerto donde “enlazar” (`bind()`) el socket UDP con el cual se intercambiarán los paquetes con OpenVPN.

Construye el objeto `msgudp`, el cual escuchará en conf.udpport e intercambiará paquetes UDP con OpenVPN.

Construye el objeto `msgjab`, el cual se conectará al servidor `conf.server` usando `conf.my_jid` como identificación y `conf.password` para autenticarse.

Luego invoca a `asyncore.loop()` (*main loop*) para procesar los eventos de recepción de paquetes UDP y mensajes XMPP y así realizar la función de gateway.

3.4. Modificaciones propias, originales a OpenVPN

Técnicamente las modificaciones realizadas agregan el siguiente soporte:

- OpenVPN/TCP/IPv6 [PF_INET6]

Permite usar TCP sobre IP versión 6 como protocolo de transporte entre procesos OpenVPN.

- OpenVPN/UDP/IPv6 [PF_INET6]

  Como en el caso anterior usando UDP sobre IP versión 6.

- OpenVPN/sockets-unix [PF_UNIX]

  Permite comunicar procesos OpenVPN con *gateways* locales usando conexiones exclusivamente locales (no expuestas a la red) para proveer un mayor grado de protección a dichas conexiones.

- *Multihome UDP* para IPv4 e IPv6

  Provee un algoritmo de selección de dirección origen adecuado para el caso de flujos UDP originados en un *host* con múltiples interfaces lógicas (varias direcciones IP).

- *payload conntrack*

  Seguimiento del estado de conexiones TCP para el problema de congestión acumulada TCP-sobre-TCP[19].

3.4.1. Breve resumen (REAME.IPv6 y CHANGES.IPv6)

A continuación se copian los archivos README.IPv6 y CHANGES.IPv6 que acompañan al código fuente desarrollado donde se resumen los cambios hechos y las funcionalidades agregadas:

REAME.IPv6:

```
# $Id: README.IPv6,v 1.1.10.4 2005/05/12 15:53:48 jjo Exp $ #

This README covers UDP/IPv6 v0.3.x ( --udp6 and --tcp6-xxxxxx  ) support
for openvpn-2.0.
```

```
Also, with address family "generalization" changes came local AF_UNIX
socket support.

Available under GPLv2 from
  http://www.irrigacion.gov.ar/juanjo/openvpn/

See "Examples" section below for usage.

* Working:
  - tcp6->tcp6 (AF_INET6)
    tested on GNU/Linux
  - upd6->upd6 (AF_INET6)
    tested on GNU/Linux, FreeBSD-5.3 and OpenBSD-3.6.
  - upd4->upd6 bound (show correctly mapped address) but requires
    --float (to be fixed soon)
  - unix-dgram->unix-dgram (AF_UNIX)
    useful for implementing local proxies that can take full advantage
    of POSIX filesystem permissions ( more powerfull access mechanisms
    than inet, even for localhost)
  - multihome (MH)
    IPv4: compiles and works OK GNU/Linux (tested)
    IPv6: compiles on GNU/Linux, should work unless it fails :o)

* Setup:
  ./configure --disable-ipv6        (enabled by default)
  ./configure --enable-unix-sockets (disabled by default)
  :

* Usage:
  For IPv6 just specify "-p upd6" an proper IPv6 hostnames, taking the
example
  from man page ...

  On may:
    openvpn --proto udp6 --remote <june_IPv6_addr> --dev tun1 --ifconfig
10.4.0.1 10.4.0.2
    --verb 5 --secret key

  On june:
    openvpn --proto udp6 --remote <may_IPv6_addr>  --dev tun1 --ifconfig
10.4.0.2 10.4.0.1
    --verb 5 --secret key

  Same for --proto tcp6-client, tcp6-server

* Examples: some succesfully tested command lines
  [  BTW did you know that openvpn can succesfully negotiate to self
     with --remote localhost ? (VERY useful for fast testing)         ]

  - IPv6 "normal" usage (+succesfully tested tunnel traffic)
    server# openvpn --proto udp6 ...
      :
      Thu Sep 23 22:15:48 2004 Peer Connection Initiated with
[AF_INET6]fe80::205:5dff:fef1:1ceb%wlan0wds1:5000
      :
```

```
    client# openvpn --proto udp6 --remote fe80::240:5ff:feae:c851 ...
      :
      Thu Sep 23 22:13:19 2004 Peer Connection Initiated with
[AF_INET6]fe80::240:5ff:feae:c851%wlan0wds0:5000
      :

  - IPv6 server, IPv4 client (more detailed)
    server# openvpn --proto udp6 ...
      :
      Thu Sep 23 22:28:36 2004 UDPv6 link local (bound):
[AF_INET6][undef]:5000
      Thu Sep 23 22:28:36 2004 UDPv6 link remote: [AF_INET6][undef]
      Thu Sep 23 22:28:50 2004 Peer Connection Initiated with
[AF_INET6]::ffff:10.55.14.253:5000
      Thu Sep 23 22:28:51 2004 Initialization Sequence Completed
      Thu Sep 23 22:28:56 2004 WARNING: Actual Remote Options ('...
proto UDPv4 ... ') \
                               are inconsistent with Expected Remote
Options ('... proto UDPv6 ...')

    client# openvpn  --remote 10.55.14.254 ...  ### same default as now:
--udp
      :
      Thu Sep 23 22:26:11 2004 UDPv4 link local (bound):
[AF_INET][undef]:5000
      Thu Sep 23 22:26:11 2004 UDPv4 link remote:
[AF_INET]10.55.14.254:5000
      Thu Sep 23 22:26:21 2004 Peer Connection Initiated with
[AF_INET]10.55.14.254:5000
      Thu Sep 23 22:26:21 2004 WARNING: Actual Remote Options ('...
proto UDPv6 ...') \
                               are inconsistent with Expected Remote
Options ('... proto UDPv4 ...')
      Thu Sep 23 22:26:22 2004 Initialization Sequence Completed

  - IPv6 loopback
    alone# openvpn --proto udp6 --remote ::1 ...
      :
      Wed Sep 22 13:03:07 2004 Peer Connection Initiated with
[AF_INET6]::1:5000
      :

  - AF_UNIX toself
    alone# openvpn --proto unix-dgram --local /tmp/o.s --remote /tmp/o.s
--dev tun  ...
      :
      Thu Sep 23 16:37:27 2004 Peer Connection Initiated with
[AF_UNIX]/tmp/o.s
      :

  - AF_UNIX between to diff instances
    peer1# openvpn --proto unix-dgram --local /tmp/o1.s --remote
/tmp/o2.s
    peer2# openvpn --proto unix-dgram --local /tmp/o2.s --remote
/tmp/o1.s
```

```
      :
      Wed Sep 22 12:49:03 2004 Peer Connection Initiated with
[AF_UNIX]/tmp/o1.s
      :


* Main code changes summary:
  - New struct openvpn_sockaddr type (hold sockaddrs and pktinfo):

    struct openvpn_sockaddr {
            union {
                    struct sockaddr sa;
                    struct sockaddr_in in;
    #ifdef USE_PF_INET6
                    struct sockaddr_in6 in6;
    #endif
    #ifdef USE_PF_UNIX
                    struct sockaddr_un un;
    #endif
            } addr;
    #if ENABLE_IP_PKTINFO
            struct in_pktinfo pi;
    #endif
    };

    struct link_socket_addr
    {
            struct openvpn_sockaddr local;
            struct openvpn_sockaddr remote;
            struct openvpn_sockaddr actual;
    };
    allows simple type overloading: local.addr, local.in, local.in6 ...
etc

  - several function prototypes moved from
    sockaddr_in to sockaddr args type
  - several new sockaddr functions needed to "generalize" AF_xxxx
operations:
    addr_copy(), addr_zero(), ...etc
    proto_is_udp(), proto_is_dgram(), proto_is_net()

* TODO: (d: done, !: fundamental, w: wanted, n: nah ... not critical, ?:
need more thought)
 --
  [d]-  ./configure [ --disable-ipv6 ] [ --enable-unix-sockets ]
        map to USE_PF_INET6 and USE_PF_UNIX
  [d]-  merge MH patch
  [d]-  -p tcp6-client, -p tcp6-server
  [d]-  MH IPv6 support
 --

  [!]-  Implement comparison for mapped addesses: server in dual stack
listening
        IPv6 must permit incoming streams from allowed IPv4 peer (ie
without --float).
```

```
  [!]-  IPv6 with actual host resolution, currently only numerical
(AI_NUMERICHOST)
  [n]-  call socket() lately, after getaddrinfo() to decide IPv4 or IPv6
host
        (hence socket()) instead of needing -p {udp|udp6}
        NOT ACTUALLY a big trouble, given that you _do_ setup both sides
        (keys, certs, etc), using udp or udp6 is actually _another_
setup bit.
  [?]-  integrate both IPv4 and IPv6 addr resolution with getaddrinfo
instead of
        venerable gethostbyname&friends, problem: horizontal portability
(across
        platforms) and vertical portab. (across versions)

--
JuanJo Ciarlante   jjo|at|mendoza.gov.ar
:                                                                      :
.                                          Linux IP Aliasing author .
.   Modular algo (AES et all) support for FreeSWAN/OpenSWAN author .
:...       plus  other scattered free software bits in the wild ...:
```

CHANGES.IPv6:

```
# $Id: CHANGES.IPv6,v 1.1.10.13 2005/05/18 19:54:05 jjo Exp $ #
* v0.3.11
  . woaooHOO: fixed udp6 MULTI (TLS server) !
* v0.3.10
  . stupid typo .... GRR

* v0.3.9
  . some MH code reorg., allow compilation with ./configure --disable-
multihome

* v0.3.8
  . udp6 --multihome (MH) support fixed, tested OK! on GNU/Linux

* v0.3.7
  . udp6 MH support: compiles, not tested.

* v0.3.6
  . tested UDPv4 MH on GNU/Linux: works ok
  . fix incorrect addr printing in print_link_sockaddr()

* v0.3.5
  . internals: kill print_link_sockaddr_ex(), just use print_propiate
flags
   (just ~10lines change at all !)
```

```
* v0.3.4
  . make tcp4-client work against tcp6-server

* v0.3.3
  . freebsd: compute true addrlen for sendto() with af_addr_size()

* v0.3.2
  . minor changes to socket.[ch] (MH merge)

* v0.3.1
  . syshead.h MH changes were missing ; now it actually compiles MH
support

* v0.3.0
  - tcp6-client, tcp6-server
  - MH patch included by default from now on

* v0.2.4-MH-0.0.6
  - account for !AF_INET in addr_host()
  - removed S_IN, S_IN6 and S_UN casts; migrate last functions to
openvpn_sockaddr: print*sockaddr*
  - more openvpn_sockaddr migration (polishing), almost ready
  - 3rd MH integration round

* v0.2.4
  - fix --route usage for udp6 (redirect-default-gateway semantics)

* v0.2.3
  - udp6 "correct" support for freebsd and openbsd
    cc and tested OK: freebsd-5.3,openbsd-3.6 against GNU/Linux

* v0.2.2
  - IPv6 (--proto udp6), unix-socket support selectable at configure-
time
    (all 4 combinations tested)
    ./configure --disable-ipv6        (enabled by default)
    ./configure --enable-unix-sockets (disabled by default)
    (internal) USE_PF_INET6, USE_PF_UNIX from autoconf
  - Change PROTO_x from #define to enum, to allow easier/cleaner support
for
    optional protocols
  - Added IPV6_xxxx_HEADER_SIZE

* v0.2.1
  First public release, see README.IPv6

# vim: sw=2
```

3.4.2. Consideraciones de portabilidad del código fuente

Las modificaciones realizadas a OpenVPN se compilaron y probaron en

las siguientes plataformas:

- Linux-2.6, glibc-2.2+, gcc-3 y gcc-4 (**nativo)**
- FreeBSD-5.3 (**emulación x86)**
- OpenBSD-3.6 (**emulación x86)**

Para la emulación x86 se utilizó QEMU [20] por sus excelentes cualidades técnicas: emulación completa y eficiente, excelente soporte de networking, uso desde línea de comandos, licencia GPL. En particular el soporte de puertos serie fue de extrema utilidad para correr los sistemas operativos emulados (*guest)* desde consolas seriales virtuales en modo texto.

La única funcionalidad que restó portar a *x*BSD (FreeBSD, OpenBSD, etc) es el soporte de *Multihome* (varias interfaces hacia la red cada una con una dirección lógica diferente) dado que el interfaz de programación para consultar/fijar la dirección IPv4/IPv6 de los paquetes UDP no es portable entre Linux y *x*BSD.

3.4.3. Generalización de la familia de *socket* de conexión

Tal como está desarrollado *OXG*, no se requieren modificaciones al código fuente de OpenVPN para utilizarlo debido a que OpenVPN y *OXG* se comunican usando UDP, soportado en forma nativa por OpenVPN.

Sin embargo, para el caso típico en que OpenVPN y *OXG* se ejecutan en el mismo host el uso de UDP como “canal” de intercambio deja expuesto el *socket* UDP atendido por *OXG* a ataques tales como denegación de servicio, captura de paquetes, etc. La familia de sockets[21] `PF_UNIX` provee canales locales de alto ancho de banda tanto `SOCK_STREAM` (flujos orientados a conexión) como `SOCK_DGRAM` (flujos de datagramas no

orientados a conexión), permitiendo asimismo **utilizar todos los mecanismos de permisos de los sistemas de archivo POSIX** (*Portable Operating System Interface*)[1] **para su acceso**, *POSIX ACLs* (*Access Control Lists*: listas de control de acceso) [22] o incluso *SELinux (Security Enhanced Linux*: mejoras de seguridad para Linux) con sus mecanismos de *MAC* (*Mandatory Access Control*: control de acceso obligatorio).

Con este objetivo en mente se modificó el código fuente de OpenVPN para agregarle capacidad de conexión vía sockets `PF_UNIX` además de `PF_INET`. Este cambio requirió hacer modificaciones estructurales al núcleo de manejo de *sockets* de OpenVPN.

Luego de realizadas dichas modificaciones, para poner a prueba la "pluralización" del soporte de diversidad de familias de protocolos (`PF_`*xxxx*) se agregó soporte para la `PF_INET6` (*sockets* IPv6).

Es interesante recalcar que esto último, el **soporte de IPv6 para transporte de datagramas OpenVPN desarrollado como parte del presente trabajo, fue la principal razón de incorporación de las modificaciones al repositorio de desarrollo oficial del proyecto**.

El nuevo escenario que plantea el despliegue de IPv6 requiere una revisión del modo de uso de la popular interfaz de programación BSD sockets. La razón fundamental de ello es que la "convivencia" (en cuanto a coexistencia y similitud) de IPv4 e IPv6 a nivel de datagrama también se ve reflejada en la interfaz de programación por las siguientes razones:

- Portabilidad del fuente: contemplar la posibilidad compilarse para en

1 *POSIX* estandariza varias interfaces de sistema operativo: programación (procesos, hilos, sistema de archivo, red, etc), línea de comando y otros. En general se lo puede ver como la estandarización de un sistema "tipo *UNIX"*.

entornos sin soporte *dual stack* (IPv4+IPv6) en archivos de cabecera y/o bibliotecas minimizando el uso de condicionales (#ifdef, etc.).

- Portabilidad horizontal entre distintos sistemas operativos , tanto "sintácticamente" (compilación correcta) como semánticamente (algo más complicado debido a que las implementaciones de IPv6 aún están en un proceso de "maduración").

- Tiempo de ejecución: es posible y totalmente legítimo que un "ejecutable" o "binario" haya sido compilado con soporte *dual stack* (capaz de usar IPv4 y/o IPv6) pero en momento de ejecutarse el núcleo del sistema operativo no tenga configurado/cargado el soporte para IPv6[1].

- DNS: la resolución directa de direcciones IPv6 usan un R.R. (*resource record*) **AAAA** y la interfaz de programación permite la posibilidad de preguntar tanto por registros **A** y/o **AAAA de un mismo nombre** usando la llamada de biblioteca `getaddrinfo(3)`.
  Notar que dado que es posible que no se conozca a priori si se trata de una dirección IPv4 ó IPv6 la elección `PF_INET` ó `PF_INET6` para la creación del `socket(2)` **podría depender del resultado devuelto por el DNS**.

3.4.3.1. Nuevo tipo de datos para sockets: struct openvpn_sockaddr (socket.h)

Debido a que el código fuente de OpenVPN asume que el *socket* que utilizará siempre es AF_INET (IPv4) fue necesario modificarlo para "generalizar" el tipo de dato utilizado.

1 Linux, por ejemplo, incorpora el soporte de IPv6 en el núcleo de forma modular (módulo **ipv6**); lo cual facilita aún más hacer su carga en forma opcional.

Se creó un nuevo tipo de datos (`struct` **`openvpn_sockaddr`**) para representar un socket generalizado tal como se muestra en el siguiente extracto de código:

```
/*
 * core socket struct: AF address and MH packet_info
 */
struct openvpn_sockaddr {
   union {
            struct sockaddr sa;       /* PF_UNSPEC */
            struct sockaddr_in in;    /* PF_INET   */
#ifdef USE_PF_INET6
            struct sockaddr_in6 in6;  /* PF_INET6  */
#endif
#ifdef USE_PF_UNIX
            struct sockaddr_un un;    /* PF_UNIX   */
#endif
   } addr;
#if ENABLE_IP_PKTINFO
   union {
            struct in_pktinfo in;     /* MH PF_INET  */
#ifdef USE_PF_INET6
            struct in6_pktinfo in6;   /* MH PF_INET6 */
#endif
   } pi;   /* Multihome support for UDP */
#endif
};
/*
 * link_socket_addr holds both endpoints sockaddrs and
 * the actual one used for packet exchange (MH)
 */
struct link_socket_addr
{
        struct openvpn_sockaddr local;
        struct openvpn_sockaddr remote;
        struct openvpn_sockaddr actual;
};
```

Notar el uso de las construcciones de preprocesador `#ifdef USE_PF_`*`xxxx`* con el objeto de optimizar espacio (y ciclos de CPU) si se compila OpenVPN sin soporte de `PF_INET6` ó `PF_UNIX`. Este hecho **no es menor,** es una muestra más de la versatilidad y optimización posible si se dispone del código fuente para construir una aplicación.

El tipo `struct openvpn_sockaddr` contiene dos `union`s:

- `union { ... } addr;`

  Contiene la dirección de un *endpoint* (*socket*), contempla todas las nuevas familias de protocolos que se adicionaron a OpenVPN.
- `union { ... } pi;`

  Contiene la información de soporte para *multihoming*: memoriza los datos de interfaz de arribo, etc. necesarios para escribirlos en los **paquetes UDP** de respuesta. Sin este cambio, OpenVPN respondería siempre desde la IP origen que establezca la tabla de *routing* hacia el la dirección del extremo remoto, siendo que un *host multihomed* **debería responder desde la dirección IP por la cual fue contactado** originalmente.

La utilización de `union` permite construcciones “polimórficas” en lenguaje C sin abusar de moldes (*casting*) y agregando claridad al código fuente, por ejemplo:

```
struct openvpn_sockaddr *o_sa;
:
switch(o_sa->addr.sa.sa_family) {
  case AF_INET:
     do_something_ipv4(&o_sa->addr.in);
     break;
  case AF_INET6:
     do_something_ipv6(&o_sa->addr.in6);
     break;
  case AF_UNIX:
     do_something_unix(&o_sa->addr.un);
 };
```

Evidentemente el cambio del tipo fundamental de dato que representa a las conexiones/asociaciones tiene un impacto importante en el resto del código fuente.

Se crearon nuevas funciones de “apoyo” a esta generalización; por

ejemplo, con el anterior tipo único `struct sockaddr_in` la copia se realizaba con una asignación simple tal como la mostrada a continuación:

```
    struct sockaddr_in *src, *dst;
    :
    dst->sin_addr=src->sin_addr;  /* copy AF_INET host addrs */
```

Para implementar la misma semántica usando el nuevo tipo de dato `struct openvpn_sockaddr` se creó la función `addr_copy_host()` mostrada a continuación:

```
    /*
     * copy host addrs based on AF
     */
    static inline void
    addr_copy_host(struct openvpn_sockaddr *dst, const struct
        openvpn_sockaddr *src)
    {
      switch(src->addr.sa.sa_family) {
        case AF_INET:
          dst->addr.in.sin_addr.s_addr = src->addr.in.sin_addr.s_addr;
          break;
#ifdef USE_PF_UNIX
     case AF_UNIX:
       strncpynt(dst->addr.un.sun_path, src->addr.un.sun_path, sizeof
dst->addr.un.sun_path);
       break;
#endif
#ifdef USE_PF_INET6
     case AF_INET6:
       dst->addr.in6.sin6_addr = src->addr.in6.sin6_addr;
       break;
#endif
   }
}
```

3.4.3.2. Generalización de las características (semántica) de los diversos protocolos (socket.h)

Otro aspecto interesante del cambio realizado fue la generalización de la s**emántica de *STREAM* ó *DGRAM*** (orientado a conexión u orientado a

paquete)***, net o local*** (protocolo de red ó mecanismo de conexión local)***,*** para las diversas familias de protocolos, tal como se muestra en la Tabla 3:

proto	***proto_name***	***is_dgram***	***is_net***	***proto_af***
proto-uninitialized	proto-NONE	0	0	AF_UNSPEC
udp	UDPv4	1	1	AF_INET
tcp-server	TCPv4_SERVER	0	1	AF_INET
tcp-client	TCPv4_CLIENT	0	1	AF_INET
udp6	UDPv6	1	1	AF_INET6
tcp6-server	TCPv6_SERVER	0	1	AF_INET6
tcp6-client	TCPv6_CLIENT	0	1	AF_INET6
unix-dgram	UNIX_DGRAM	1	0	AF_UNIX
unix-stream	UNIX_STREAM	0	0	AF_UNIX

Tabla 3: Nombres de protocolo y generalización de las características de los mismos

Lo anterior, expresado en el código fuente (socket.h):

```
  struct proto_names {
    const char *short_form;
    const char *display_form;
    bool is_dgram;
    bool is_net;
    sa_family_t proto_af;
  };

  /* Indexed by PROTO_x */
  static const struct proto_names proto_names[PROTO_N] = {
    {"proto-uninitialized",        "proto-NONE",0,0, AF_UNSPEC},
    {"udp",         "UDPv4",1,1, AF_INET},
    {"tcp-server", "TCPv4_SERVER",0,1, AF_INET},
    {"tcp-client", "TCPv4_CLIENT",0,1, AF_INET},
    {"tcp",         "TCPv4",0,1, AF_INET},
  #ifdef USE_PF_INET6
    {"udp6"       ,"UDPv6",1,1, AF_INET6},
    {"tcp6-server","TCPv6_SERVER",0,1, AF_INET6},
    {"tcp6-client","TCPv6_CLIENT",0,1, AF_INET6},
    {"tcp6"       ,"TCPv6",0,1, AF_INET6},
  #endif
  #ifdef USE_PF_UNIX
    {"unix-dgram" ,"UNIX_DGRAM",1,0, AF_UNIX },
    {"unix-stream","UNIX_STREAM",1,0, AF_UNIX }
  #endif
:
```

Para lograr un mejor encapsulamiento de estas “características” de los diversos protocolos se crearon funciones de apoyo tal como la mostradas a continuación:

```
bool
proto_is_net(int proto)
{
  if (proto < 0 || proto >= PROTO_N)
    ASSERT(0);
  return proto_names[proto].is_net;
}
bool
proto_is_dgram(int proto)
{
  if (proto < 0 || proto >= PROTO_N)
    ASSERT(0);
  return proto_names[proto].is_dgram;
}
bool
proto_is_udp(int proto)
{
  if (proto < 0 || proto >= PROTO_N)
    ASSERT(0);
  return proto_names[proto].is_dgram&&proto_names[proto].is_net;
}
bool
proto_is_tcp(int proto)
{
  if (proto < 0 || proto >= PROTO_N)
    ASSERT(0);
  return (!proto_names[proto].is_dgram)&&proto_names[proto].is_net;
}
sa_family_t
proto_sa_family(int proto)
{
  if (proto < 0 || proto >= PROTO_N)
    ASSERT(0);
  return proto_names[proto].proto_af;
}
```

Un extracto de código que utiliza estas nuevas funciones se muestra a continuación (*unified diff*):

```
      return res;
    }
-  else if (sock->info.proto == PROTO_TCPv4_SERVER || sock->info.proto
== PROTO_TCPv4_CLIENT)
```

```
+  else if (proto_is_tcp(sock->info.proto)) /* unified TCPv4 and TCPv6
*/
     {
       /* from address was returned by accept */
```

3.4.3.3. Soporte IPv6 y PF_UNIX: nuevas opciones para autoconf (configure.ac)

OpenVPN utiliza autoconf (./configure) para preparar el entorno y opciones de compilación en las diversas plataformas soportadas. Para permitir la compilación de openvpn con los nuevos tipos de *sockets* se agregaron las siguientes opciones:

```
./configure --disable-ipv6         (enabled by default)
./configure --enable-unix-sockets  (disabled by default)
```

Con el correspondiente agregado al código fuente (archivo `configure.ac`):

```
AC_ARG_ENABLE(ipv6,
  [  --disable-ipv6          Disable UDP/IPv6 support],
  [PF_INET6="$enableval"],
  [PF_INET6="yes"]
)

AC_ARG_ENABLE(unix-sockets,
  [  --enable-unix-sockets   Enable PF_UNIX sockets links],
  [PF_UNIX="$enableval"],
  [PF_UNIX="no"]
)

:...
```

```
  if test "$PF_UNIX" = "yes"; then
    AC_CHECKING([for sys/un.h header file for PF_UNIX])
    AC_CHECK_HEADER(sys/un.h,
        [AC_DEFINE(USE_PF_UNIX, 1, [Compile support for PF_UNIX
sockets])],
        [AC_MSG_ERROR([sys/un.h header not found.])]
    )
  fi

  if test "$PF_INET6" = "yes"; then
   AC_CHECKING([for struct sockaddr_in6 for IPv6 support])
   AC_CHECK_TYPE(
       [struct sockaddr_in6],
       [AC_DEFINE(USE_PF_INET6, 1, [struct sockaddr_in6 is needed for
IPv6 peer support])],
       [],
       [#include "syshead.h"])
  fi

  :
```

3.4.4. Soporte *Multihome (UDP)*

Dado que UDP es no orientado a conexión, la dirección IP[1] origen presente en el flujo de datagramas de "respuesta" de un host remoto **no necesariamente es igual a la IP dirección destino** con la cual se contactó al mismo **si el host remoto posee varias direcciones IP (*multihome*).** OpenVPN tiene como política negar la conexión si dichas direcciones IP no coinciden.

El soporte ***Multihome*** permite a la máquina que responde emitir sus datagramas desde el mismo IP con el que fue contactado en vez de dejar dicha decisión librada a lo que la configuración de *routing*[2] dicte.

Es importante destacar que esta parte del desarrollo se basó fuertemente

1 Válido tanto para IPv4 como IPv6.

2 La dirección IP origen de los datagramas que se generen desde un host hacia un destino/mask **es** uno de los campos de las entradas de la tabla de *routing* (en Linux se puede ver como "src" al ejecutando "ip route")

en el realizado previamente para IPv4 por James Yonan, el autor de OpenVPN.

3.4.4.1. Soporte Multihome: cambios a struct openvpn_sockaddr (socket.h)

El siguiente extracto de código muestra el agregado del nuevo `union { ... } pi` que contendrá los datos relevantes del último datagrama recibido:

```
#if ENABLE_IP_PKTINFO
     union {
             struct in_pktinfo in;     /* MH PF_INET  */
  #ifdef USE_PF_INET6
             struct in6_pktinfo in6;   /* MH PF_INET6 */
  #endif
     } pi;   /* Multihome support for UDP */
  #endif
```

Donde `struct in_pktinfo` está declarada de la siguiente manera:

```
struct in_pktinfo {
    unsigned int   ipi_ifindex;  /* Interface index */
    struct in_addr ipi_spec_dst; /* Local address */
    struct in_addr ipi_addr;     /* Header Destination address */
};
```

donde

- `int ipi_ifindex`

  Es el número (índice) de la interfaz de red por la cual se recibió el paquete

- `struct in_addr ipi_spec_dst`

Es la dirección IP local que se le quiere dar al paquete, pasando `IP_PKTINFO` a `sendmsg(2)`.

- `struct in_addr ipi_addr`

  Es la dirección IP destino presente en el paquete recibido.

Los datos para IPv6 son análogos.

3.4.4.2. Soporte Multihome: cambios las funciones de recepción y transmisión de datagramas (socket.c)

A continuación se muestra un extracto de código fuente donde se le pide al *kernel* (núcleo del sistema operativo) que se desea dicha información adicional:

```
#if ENABLE_IP_PKTINFO
  else if (flags & SF_USE_IP_PKTINFO) /* runtime flag */
    {
      int pad = 1;
      setsockopt (sd, SOL_IP, IP_PKTINFO, (void*)&pad, sizeof(pad));
    }
#endif
```

Asimismo donde se hace uso de la misma en momento de recepción del datagrama:

```
#if ENABLE_IP_PKTINFO
/* UDPv4 and UDPv6 */
static socklen_t
link_socket_read_udp_posix_recvmsg (struct link_socket *sock,
                                    struct buffer *buf,
                                    int maxsize,
                                    struct openvpn_sockaddr *from)
{
  struct iovec iov;
  union openvpn_pktinfo opi;
  struct msghdr mesg;
  socklen_t fromlen = sizeof (from->addr);
```

```
    iov.iov_base = BPTR (buf);
    iov.iov_len = maxsize;
    mesg.msg_iov = &iov;
    mesg.msg_iovlen = 1;
    mesg.msg_name = &from->addr;
    mesg.msg_namelen = fromlen;
    mesg.msg_control = &opi;
    mesg.msg_controllen = sizeof (opi);
    buf->len = recvmsg (sock->sd, &mesg, 0);
    if (buf->len >= 0)
      {
        struct cmsghdr *cmsg;
        fromlen = mesg.msg_namelen;
        cmsg = CMSG_FIRSTHDR (&mesg);
        if (cmsg != NULL
           && CMSG_NXTHDR (&mesg, cmsg) == NULL
           && cmsg->cmsg_level == SOL_IP
           && cmsg->cmsg_type == IP_PKTINFO
           && cmsg->cmsg_len >= sizeof (struct openvpn_in_pktinfo))
         {
           struct in_pktinfo *pkti = (struct in_pktinfo *) CMSG_DATA
(cmsg);
           from->pi.in.ipi_ifindex = pkti->ipi_ifindex;
           from->pi.in.ipi_spec_dst = pkti->ipi_spec_dst;
         }
  #ifdef USE_PF_INET6
        else if (cmsg != NULL
           && CMSG_NXTHDR (&mesg, cmsg) == NULL
           && cmsg->cmsg_level == IPPROTO_IPV6
           && cmsg->cmsg_type == IPV6_PKTINFO
           && cmsg->cmsg_len >= sizeof (struct openvpn_in6_pktinfo))
         {
           struct in6_pktinfo *pkti6 = (struct in6_pktinfo *) CMSG_DATA
(cmsg);
           from->pi.in6.ipi6_ifindex = pkti6->ipi6_ifindex;
           from->pi.in6.ipi6_addr = pkti6->ipi6_addr;
         }
  #endif
      }
    return fromlen;
  }
  #endif
```

Y finalmente donde se aplica la dirección al datagrama saliente:

```
#if ENABLE_IP_PKTINFO
int
link_socket_write_udp_posix_sendmsg (struct link_socket *sock,
                                     struct buffer *buf,
                                     struct openvpn_sockaddr *to)
{
```

```
  struct iovec iov;
  struct msghdr mesg;
  struct cmsghdr *cmsg;

  /* ASSERT(sock->info.lsa->remote.addr.in.sin_family == AF_INET); */
  iov.iov_base = BPTR (buf);
  iov.iov_len = BLEN (buf);
  mesg.msg_iov = &iov;
  mesg.msg_iovlen = 1;
  switch (sock->info.lsa->remote.addr.sa.sa_family) {
    case AF_INET: {
       struct openvpn_in_pktinfo opi;
       struct in_pktinfo *pkti;
       mesg.msg_name = &to->addr.sa;
       mesg.msg_namelen = sizeof (struct sockaddr_in);
       mesg.msg_control = &opi;
       mesg.msg_controllen = sizeof (opi);
       mesg.msg_flags = 0;
       cmsg = CMSG_FIRSTHDR (&mesg);
       cmsg->cmsg_len = sizeof (opi);
       cmsg->cmsg_level = SOL_IP;
       cmsg->cmsg_type = IP_PKTINFO;
       pkti = (struct in_pktinfo *) CMSG_DATA (cmsg);
       pkti->ipi_ifindex = to->pi.in.ipi_ifindex;
       pkti->ipi_spec_dst = to->pi.in.ipi_spec_dst;
       pkti->ipi_addr.s_addr = 0;
       break;
    }
#ifdef USE_PF_INET6
    case AF_INET6: {
       struct openvpn_in6_pktinfo opi6;
       struct in6_pktinfo *pkti6;
       mesg.msg_name = &to->addr.sa;
       mesg.msg_namelen = sizeof (struct sockaddr_in6);
       mesg.msg_control = &opi6;
       mesg.msg_controllen = sizeof (opi6);
       mesg.msg_flags = 0;
       cmsg = CMSG_FIRSTHDR (&mesg);
       cmsg->cmsg_len = sizeof (opi6);
       cmsg->cmsg_level = IPPROTO_IPV6;
       cmsg->cmsg_type = IPV6_PKTINFO;
       pkti6 = (struct in6_pktinfo *) CMSG_DATA (cmsg);
       pkti6->ipi6_ifindex = to->pi.in6.ipi6_ifindex;
       pkti6->ipi6_addr = to->pi.in6.ipi6_addr;
       break;
    }
#endif
    default: ASSERT(0);
  }
  return sendmsg (sock->sd, &mesg, 0);
}
#endif
```

3.4.4.3. Soporte Multihome: adiciones a autoconf (configure.ac)

Para soportar la configuración opcional del soporte *multihome* en tiempo de compilación:

```
./configure --disable-multihome     (enabled by default)
```

Se agregaron las siguientes líneas a `configure.ac`:

```
AC_ARG_ENABLE(multihome,
   [  --disable-multihome     Disable multi-homed UDP server support (--
multihome)],
   [MULTIHOME="$enableval"],
   [MULTIHOME="yes"]
)

:
:

dnl compile --multihome option
if test "$MULTIHOME" = "yes"; then
   AC_DEFINE(ENABLE_MULTIHOME, 1, [Enable multi-homed UDP server
capability])
fi
```

3.4.5. Optimización para el caso TCP-sobre-TCP: *payload-conntrack*

Es conocido el inconveniente que presentan las *VPN*s que utilizan TCP para vincularse cuando éstas transportan conexiones TCP[19]: frente a congestión ó pérdida de segmentos el TCP de la capa superior genera retransmisiones que son almacenadas en los *buffers* (memoria intermedia) del TCP de la capa inferior para así garantizar su entrega. Siendo que se trata de retransmisiones, sería **suficiente entregar en forma confiable el primer segmento** ignorando --filtrando-- las repeticiones del mismo.

Precisamente este “filtrado” es lo que implementa la optimización desarrollada.

Notar que el uso de *OXG* adolece del mismo problema (TCP-sobre-TCP) cuando se transporta TCP en capa superior debido a que *XMPP* usa TCP tal como se muestra en la Figura 8

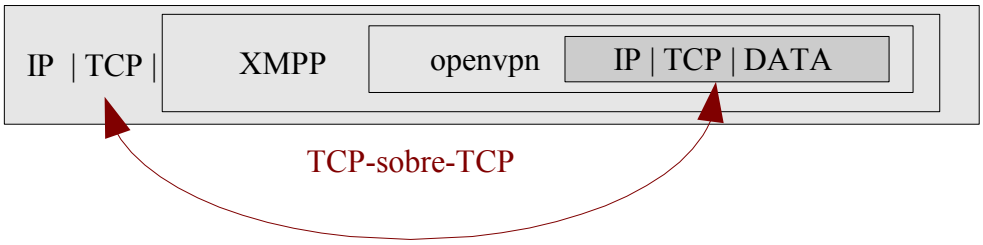

Figura 8: OpenVPN sobre XMPP también **es** TCP-sobre-TCP

La modificación que implementa la optimización mediante el filtrado de las retransmisiones TCP se denominó **payload conntrack** dado que en efecto utiliza los mismos mecanismos que un *firewall statefull* (dispositivo de filtrado de paquetes basado en estado de las conexiones), al **memorizar el estado de cada conexión**.

El filtrado de retransmisiones TCP tiene que ser MUY cuidadoso de no suprimir simplemente aquellos segmentos TCP que tuvieren los mismos números de secuencia de transmisión `[SEG.SEQ]`[1] y de acuse de recibo `[SEQ.ACK]` como dictaría el sentido común debido a que TCP posee **diversas semánticas asociadas a la retransmisión** que no necesariamente corresponden a “datos repetidos”:

- *Duplicated ACKs* (acuses de recibo duplicados) (RFC2581) [23]

  Aparecen cuando en el lado TCP *RX* (receptor) se ha producido un “hueco” en el espacio de secuencia pero se han recibido correctamente

1 Notación de acuerdo a RFCs.

segmentos “posteriores”: el TCP *RX* envía segmentos sin “datos” y sin avanzar el *ACK.*
En principio se podría suponer que dichos segmentos son redundantes e innecesarios, pero el TCP *TX* activa el mecanismo heurístico denominado *duplicated ACKs* para inferir que se ha producido el mencionado “hueco” procediendo entonces a enviar el segmento correspondiente.

- *Zero window probe* (prueba de ventana = 0) (RFC1122 - Sec.4.2.2.17) [24]

  Cuando TCP *RX* “cierra” la ventana de recepción `[RCV.WND=0]` el lado *TX* (transmisor) debería activar la prueba de “ventana cero”, típicamente implementada enviando un segmento duplicado **con 1 sólo byte** de datos (el último byte confirmado positivamente por el lado receptor[1]) para forzar al lado *RX* a enviar el *ACK* correspondiente y así obtener una actualización del valor de ventana de recepción `[RCV.WND]`.
  Obviamente estos segmentos “retransmitidos” tampoco deben ser filtrados.

- TCP *selective ACKs option (SACKs)* (RFC2018) [25]

  El uso de *selective ACKs* (acuses de recibo explícitos por rango) efectivamente amplía la información de espacio de secuencia recibido correctamente y, al ser una opción del encabezado TCP, obviamente deber ser considerada como una “continuación” del mismo.

- TCP *window scale option (WSCALE)* (RFC1323) [26]

1 Informalmente diríamos “el último byte *ACK*-eado”.

Esta opción de TCP cambia la interpretación del campo TCP.WINDOW y se debe tratar con el mismo cuidado que las anteriores.

3.4.5.1. Payload conntrack: tabla de estado de conexiones (payload.h: interfaz pública)

- `struct payload_context;`

Existe una nueva **tabla de estado de conexiones por cada túnel de OpenVPN** representada en el nuevo `struct payload_context`. OpenVPN puede manejar varios túneles por proceso, de allí que no exista una tabla global.

```
  struct payload_context {
    struct hash *hash;
    struct schedule *schedule; /* unused by now */
    int max_tcp_conns; /* unused by now */
    int tcp_retrans;
    struct {
      int bucket_base;       /* last visited bucket */
      int buckets_per_pass;  /* how many buckets per pass
*/
      struct event_timeout wakeup; /* timer tick for this
gc */
    } gc;
  };
```

El puntero a la tabla *per se* es `struct hash *hash` , la cual es una *hashtable*[27] (tabla indexada) construída utilizando las bibliotecas nativas de OpenVPN. Se usan los datos de la *5-upla* de conexión como clave de la entrada en la *hashtable*.

Para la expiración de las entradas “antiguas” se corre un *garbage collector* (recolector de memoria no usada) desde un *timer* (temporizador) una vez por segundo. El mismo recorre `buckets_per_pass` cabezas de lista por vez, almacenando la próxima

a recorrer en `bucket_base`. El *timer* se desactiva cuando no hay ninguna entrada en la *hashtable*.

Dado que cada túnel OpenVPN guarda estado de la misma en una estructura denominada `struct context_2` , se agregó un puntero a `struct payload_context`:

```
  struct context_2 {
    :
    struct link_socket *link_socket;       /* socket used
for TCP/UDP connection to remote */
    struct openvpn_sockaddr *to_link_addr;         /* IP
address of remote */
    struct openvpn_sockaddr from;               /* address
of incoming datagram */
    :
  #ifdef USE_PAYLOAD_CONNTRACK
    struct payload_context *payload_context;
  #endif
  }
```

- `struct payload_context * payload_new(int tcp_retrans);`

  Invocada en momento de creación del túnel para crear la tabla de estados, `tcp_retrans` es el tiempo (configurable) durante el cual se aplicará el filtrado a los segmentos sucesivos retransmitidos desde el momento en que se memorizó el primero. Tiempo de vida: igual al del túnel.

- `void payload_free(struct payload_context *);`

  Destruye la tabla de estados, usada al momento de liberar los recursos asociados a un túnel (destrucción del túnel).

- `int payload_tcp_retrans_drop(struct context *c, struct buffer *buf);`

Se diseñó para minimizar los cambios en el código fuente original --lo menos “invasivo” posible--, actualmente es la única función *pública* relacionada con este módulo.

Invocada por cada datagrama que atraviesa OpenVPN.

La información de estado es memorizada en el contexto general de la conexión (`struct context`, campo `payload_context`), `struct buffer` apunta a donde se halla el contenido del datagrama.

El extracto de código donde se llama a la función:

```
#if USE_PAYLOAD_CONNTRACK
      if (c->c2.payload_context)
      {
             if (payload_tcp_retrans_drop(c, &c->c2.buf))
             {
                    buf_reset (&c->c2.to_link);
                    goto out;
             }
      }
#endif
```

Notar la simpleza de uso a pesar de la complejidad de la implementación.

3.4.5.2. Payload conntrack: filtrado de segmentos TCP (payload.h: interfaz pública)

- `struct payload_tuple_id`

Representa los datos para distinguir una única conexión.

```
/*
 * struct payload_tuple_id : uniq 5-upla (proto==TCP is implicit)
 */
struct payload_tuple_id {
  /* BEGIN uniq TCP 5-upla id */
  uint32_t ip_saddr, ip_daddr;
  uint16_t tcp_sport, tcp_dport;
  /* END   uniq TCP 5-upla id */
};
```

Dado que el desarrollo presente **solamente contempla filtrado de TCP** no es necesario el campo “protocol” de la *5-upla* de conexión/asociación, quedando los 4 campos: IP origen/destino y puertos origen/destino.

- `struct payload_tuple`

  Contiene todos los datos de estado relevantes de cada conexión.

```
/*
 * payload_tuple: 1 per TCP connection, currently only TX side hook
 */
struct payload_tuple {
  struct payload_tuple_id id;
  /* round robin array with PAYLOAD_N_TCPSEGS latest tcp segments: */
  struct {
    struct openvpn_tcphdr tcph;
    int tcp_len;
    int hits;
  } tcp_seg[PAYLOAD_N_TCPSEGS];
  int tcp_seg_idx;     /* next slot to use */
  time_t last_used;
  time_t expires;
  int conn_hits;
  int deleted;
};
```

Para una misma conexión, se memorizan `PAYLOAD_N_TCP_SEGS` encabezados TCP en el vector `tcp_seg[PAYLOAD_N_TCPSEGS]` y el largo del segmento dado que el mismo no está explícito en la cabecera TCP, estos serán usados para filtrar posteriores segmentos que coincidan con alguno de estos en su encabezado y longitud.
Se almacenan varias estampas de tiempo para expirar la conexión y para **desactivar el filtrado del segmento si ha pasado *tcp_retrans tiempo* desde que se almacenó el original**, esto tiene como objetivo aplicar el filtrado solamente en las primeras retransmisiones (cuando la

frecuencia de las mismas es más alta).

3.4.5.3. Payload conntrack: implementación (payload.c)

La implementación se puede leer del código fuente adjunto; se destacan a continuación algunos extractos del mismo:

- `static inline bool tcp_opt_process(...);`

  Función genérica de procesamiento de opciones del encabezado TCP: para cada opción de TCP presente invoca *callback* (función registrada para su invocación posterior).

```
  /*
   * Loop over TCP options, call callback() if matches passed TCP opt
(wildcard: OPENVPN_TCPOPT_ANY)
   *
   * Some examples:
   * - is SACK present?
   *   if(tcp_opt_process(buf, OPENVPN_TCPOPT_SACK, NULL, NULL, pip,
ptcp))
   * - if SACK is present, call myfunc(..., myarg, ...)
   *   if(tcp_opt_process(buf, OPENVPN_TCPOPT_SACK, myfunc, myarg, ...))
   * - is any option present? (except EOL, NOP)
   *   if(tcp_opt_process(buf, OPENVPN_TCPOPT_ANY, NULL, NULL, ...))
   * - for each option (except EOL, NOP) call myfunc(..., myarg, ...)
   *   if(tcp_opt_process(buf, OPENVPN_TCPOPT_ANY, myfunc, myarg, ...))
   *
   */
   static inline bool
      tcp_opt_process (struct buffer *buf, int optnum,
      bool (*callback)(uint8_t *opt, int optlen, void *callback_arg),
      void *callback_arg)
```

- En el presente desarrollo se utiliza para **omitir la consideración de los segmentos con SACK ó WSCALE** tal como se muestra a continuación:

```
  static inline bool
  tcp_dd_opt_skip_segment(uint8_t *opt, int optlen, void *arg)
  {
    switch(*opt)
```

```
      {
      case OPENVPN_TCPOPT_SACK:
      case OPENVPN_TCPOPT_WSCALE:
        return true;
      }
    return false;
  }
  :
        /*
         * Avoid filtering out if:
         * - SYN or FIN
         * - zero window
         * - zero window probe (data size=1)
         * - SACK or WSCALE option present
         */
        if ( (ptcp->flags & (OPENVPN_TCPH_SYN_MASK|
OPENVPN_TCPH_FIN_MASK))
            || tcph.window == 0
            || (ip_totlen-OPENVPN_TCPH_GET_DOFF(tcph.doff_res))==1
            || tcp_opt_process(buf, OPENVPN_TCPOPT_ANY,
tcp_dd_opt_skip_segment, NULL))
         {
           dmsg(D_PAYLOAD_CONNTRACK, "payload_tcp_dd_drop_hit: SKIP
segment " PAYLOAD_FMT_FULL,
                PAYLOAD_FMT_FULL_ARGS(pip, ptcp));
           goto done;
         }
```

3.4.5.4. Payload conntrack: Nuevas opciones para autoconf (configure.ac)

Para permitir la compilación de openvpn con soporte de *payload conntrack*:

```
./configure --enable-payload-conntrack (disabled by
default)
```

Con el correspondiente agregado al código fuente (archivo `configure.ac`):

```
  AC_ARG_ENABLE(payload-conntrack,
     [  --enable-payload-conntrack   Enable payload conntrack for eg.
TCP retrans. dd for reliable links],
     [PAYLOAD_CONNTRACK="$enableval"],
     [PAYLOAD_CONNTRACK="no"]
  )

  :...
  dnl enable payload-conntrack optimizations
  if test "$PAYLOAD_CONNTRACK" = "yes"; then
     AC_DEFINE(USE_PAYLOAD_CONNTRACK, 1, [Enable payload conntrack])
  fi
```

4. Resultados del uso de *OXG*

Dado que el soporte de IPv6 ya ha sido depurado, aceptado en el árbol CVS oficial y se trata de una extensión de una funcionalidad presente, remito a la lectura del archivo README.IPv6 donde se describe el modo de uso.

A continuación se presentan los resultados del uso de *OXG* como *gateway* UPD/XMPP.

4.1. Ejecución

A continuación se muestran los pasos realizados para el montaje de la VPN

- Se crearon los **archivos de configuración** en los extremos OV1 y OV2 (ver Figura 2) tal como muestra la Tabla 4; nótese la simetría de los mismos.

ov1.conf	***ov2.conf***

`#archivo ov1.conf`
`my_jid=jjo@lugmen.org.ar/`**`ov1`**
`peer_jid=jjo@lugmen.org.ar/`**`ov2`**
`password=XXXXXX`
`udpport=1194`
	`#archivo ov2.conf`
	`my_jid=jjo@lugmen.org.ar/`**`ov2`**
	`peer_jid=jjo@lugmen.org.ar/`**`ov1`**
	`password=XXXXXX`
	`udpport=1194`

Tabla 4: OXG: Archivos de configuración

- Se **arrancaron sendos *gateways OXG***, tal como muestra la Tabla 5, luego de lo cual los gateways se encuentran enlazados (*bind*) al puerto UDP 1194 de cada *host.*
  Nótese que la ejecución se realiza con desde un usuario sin privilegios de administrador (uid!=0).

OV1	*OV2*
`bash$ ./main.py ov1.conf`	`bash$ ./main.py ov2.conf`

Tabla 5. Ejecución del gateway

- Se **ejecutó el proceso openvpn** propiamente dicho en ambos *hosts* tal como se muestra en la Tabla 6, ahora sí con privilegios de administrador (necesarios para los cambios en la configuración de red que debe realizar el proceso). Nótese que el *host* remoto es “localhost” debido a que el *gateway* está corriendo en el mismo *host*.
  En la Tabla 7 se muestran los mensajes de arranque obtenidos en OV1 (*stdout, stderr*) donde se observa claramente la gestión exitosa de la conexión: “**Initialization Sequence Completed”.**

OV1	*OV2*

`bash# openvpn --dev tun \`
`--proto udp \`
`--remote localhost \`
`--lport 5020 \`
`--rport 1194 \`
`--secret openvpn.key \`
`--ifconfig 1.1.1.1 1.1.1.2`
	`bash# openvpn --dev tun \`
	`--proto udp \`
	`--remote localhost \`
	`--lport 5020 \`
	`--rport 1194 \`
	`--secret openvpn.key \`
	`--ifconfig 1.1.1.2 1.1.1.1`

Tabla 6. Ejecución del proceso openvpn

OV1: stdout, stderr de openvpn

```
Mon Apr  1 19:21:21 2005 OpenVPN 2.0_rc16 i686-pc-linux [SSL] [LZO] \
  [EPOLL] [PF_INET6] [PF_UNIX] built on Apr  4 2005
Mon Apr  1 19:21:21 2005 TUN/TAP device tun0 opened
Mon Apr  1 19:21:21 2005 /sbin/ifconfig tun0 1.1.1.2 pointopoint
1.1.1.1 mtu 1500
Mon Apr  1 19:21:21 2005 UDPv4 link local (bound):
[AF_INET][undef]:5020
Mon Apr  1 19:21:21 2005 UDPv4 link remote: [AF_INET]127.0.0.1:1194
Mon Apr  1 19:21:31 2005 Peer Connection Initiated with [AF_INET]\
  127.0.0.1:1194
Mon Apr  1 19:21:32 2005 Initialization Sequence Completed
```

Tabla 7. Salida por pantalla del proceso openvpn de OV1

4.2. Mediciones

Una vez establecida la *VPN* se procedió a utilizar las herramientas estándar de diagnóstico TCP/IP.

Para cada herramienta se muestra el resultado de las mediciones vía *gateway OXG* vs. "nativo". Se tomó especial cuidado para **que el "camino" de los mensajes *XMPP* vs paquetes UDP nativos fuera exactamente el mismo**.

Las direcciones IP utilizadas para el extremo remoto son 1.1.1.2 para el vínculo vía *gateway* y 192.168.2.12 para el vínculo "nativo".

4.2.1. ping

En la Tabla 8 se observan los valores obtenidos del comando "ping" (10 paquetes)

Es interesante analizar las diferencias:

- **rtt avg**: 80.982ms vs 23.326ms

  Muy baja sobrecarga de encapsulamiento y procesamiento teniendo en cuenta que se han sumado 3 procesos a la “transformación” del flujo:

  --gateway--servidor_Jabber--gateway--

- **rtt mdev**: 62.389ms vs 3.950ms

  Aquí sí se puede ver una diferencia importante en la desviación estándar provocada posiblemente por el *buffering* de *XMPP* sobre TCP

- **packet loss:** 0% vs 10%

  Dado que el vínculo subyacente es *XMPP*/TCP/IP no hay pérdida de paquetes para el 1er caso; éste fenómeno no resulta conveniente como podría suponerse a primera vista debido al problema del “apilamiento” TCP-sobre-TCP [19]descripto anteriormente.

ping sobre VPN via gateway	*ping sobre VPN “nativo”*

`bash$ ping -q -c 10 1.1.1.2`
`PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.`
 	
`--- 1.1.1.2 ping statistics ---`
`10 packets transmitted, 10 received, 0% packet loss, time 9006ms`
`rtt min/avg/max/mdev = 50.787/80.892/264.787/62.389 ms`
	`bash$ ping -q -c 192.168.2.12`
	`PING 192.168.2.12 (192.168.2.12) 56(84) bytes of data.`
	
	`--- 192.168.2.12 ping statistics ---`
	`10 packets transmitted, 9 received, 10% packet loss, time 9008ms`
	`rtt min/avg/max/mdev = 19.596/23.326/33.147/3.950 ms`

Tabla 8. Comparación de ping

4.2.2. ping -A (adaptative ping)

La misma prueba anterior, pero con la opción **-A** puede verse en la Tabla 9. Esta opción adapta el intervalo entre paquetes al RTT tal que el próximo ECHO REQUEST es lanzado cuando se recibe el ECHO REPLY anterior (en vez de esperar 1 segundo). Este modo de operación “carga” al vínculo de tal manera que siempre exista un paquete “circulando”.

Se pueden sacar conclusiones similares al caso anterior.

ping -A sobre VPN via gateway	*ping -A sobre VPN "nativo"*
bash$ ping -A -q -c 10 1.1.1.2 PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data. --- 1.1.1.2 ping statistics --- 10 packets transmitted, 10 received, 0% packet loss, time 1807ms rtt min/avg/max/mdev = 52.604/84.087/167.336/38.497 ms, ipg/ewma 200.803/89.054 ms	bash$ ping -A -q -c 192.168.2.12 PING 192.168.2.12 (192.168.2.12) 56(84) bytes of data. --- 192.168.2.12 ping statistics --- 10 packets transmitted, 10 received, 0% packet loss, time 1809ms rtt min/avg/max/mdev = 19.243/42.194/76.596/20.330 ms, ipg/ewma 201.027/40.112 ms

Tabla 9. Comparación de ping -A

4.2.3. netperf

Netperf es una herramienta comúnmente utilizada para medir ancho de banda en transferencias TCP/IP.

En la Tabla 6 se observan los valores obtenidos

- **throughput**: 195.62 kbit/sec vs 891.15 kbit/sec
  La diferencia importante que se observa aquí no sólo es debida al encapsulamiento *per se* (con el consiguiente aumento de tamaño) sino al "apilamiento" TCP-sobre-TCP [19] descripto anteriormente.

Netperf sobre VPN via gateway	*Netperf sobre VPN "nativo"*
bash$ netperf -fk -H 1.1.1.2 TCP STREAM TEST to 1.1.1.2 Recv Send Send Socket Socket Message Elapsed Size Size Size Time Throughput bytes bytes bytes 10^3bits/sec 87380 16384 16384 195.62	bash$ netperf -fk -H 192.168.2.12 TCP STREAM TEST to 1.1.1.2 Recv Send Send Socket Socket Message Elapsed Size Size Size Time Throughput bytes bytes bytes 10^3bits/sec 87380 16384 16384 891.15

Tabla 10. *Thoughput* comparativo

5. Conclusiones

Utilizando herramientas disponibles en cualquier instalación de GNU/Linux fue posible construir un gateway que permitiera crear una VPN con **cualquier otro extremo remoto** no alcanzable directamente en la red pública (no “instalado” en el perímetro de la red), con la sola condición que las políticas de seguridad de borde permitieran pasaje de flujos de mensajería instantánea en ambos extremos.

Esto tiene una fuerte implicancia de seguridad: dado que una VPN es un mecanismo de conexión sobre el cual se puede transportar cualquier protocolo, el presente desarrollo permite el intercambio de información arbitraria desde/hacia el exterior de la organización potencialmente violando las política de seguridad de la misma.

Esto demuestra nuevamente la **relativa eficacia** que pueden prestar las políticas de seguridad basadas en parámetros del flujo (protocolo, dirección, puertos) debido a que, con las herramientas adecuadas, **cualquier flujo bidireccional de baja latencia puede ser utilizado para perforar el perímetro de seguridad.**

Las mediciones de latencia y ancho de banda del enlace obtenido muestran la factibilidad de uso de la VPN lograda para escenarios típicos tales como sesiones remotas de texto/gráficas, web/intranet, transferencia de mail.

Como resultado adicional de la presente investigación, las modificaciones realizadas a OpenVPN para el soporte de IPv6 fueron anunciadas en Septiembre de 2004 en las listas de correo *openvpn-devel* [28] para su análisis, discusión y eventual incorporación al árbol oficial de desarrollo.

En Julio de 2005 las modificaciones al código fuente de OpenVPN fueron aprobadas por el equipo de desarrollo de OpenVPN e incorporadas al repositorio de desarrollo oficial CVS [29] y futuros *snapshots* (contenido del repositorio en un determinado momento).

Las mismas estarán presentes en la futura versión 2.5.

5.1. Disponibilidad del código fuente del presente trabajo

El código fuente que acompaña el presente trabajo está disponible bajo licencia GPL[30] en http://www.um.edu.ar/pitagoras/jjo/tesis/

El mismo consiste en dos archivos en comprimidos:

- `oxg-0.1.2.tar.gz`

  (4Kb, aprox. 400 líneas de texto, lenguaje Python).

  Código fuente de OXG, ver .

  Incluído en el presente documento.

- `openvpn-2.0-udp6-jjo-v0.3.12-MH.patch.gz`

  (187Kb descomprimido, aprox 5600 líneas de texto, lenguaje C, formato *universal diff*)

  Modificaciones al código fuente de OpenVPN-2.0, ver 3.4.

  No incluído en el presente documento por razones de formato[1] y

1 Debido a que se trata de un archivo *patch* generado con *diff -u ...* es más simple leerlo directamente que disponer del mismo "pegado" en un apéndice del presente.

de espacio.

5.2. Direcciones futuras

Con las bibliotecas adecuadas es posible utilizar otros sistemas de mensajería instantánea más “populares” y por ende típicamente más “permitidos” para atravesar el perímetro de seguridad de la red. El lenguaje de programación Python y el desarrollo modular del presente trabajo permitirán realizar estas modificaciones con relativamente poco esfuerzo.

Queda aún por realizar las modificaciones a *OXG* para soportar PF_UNIX (archivos de configuración y núcleo).

El soporte de ***payload conntrack*** agregado a OpenVPN es bastante experimental en el estado en que se encuentra a la fecha, requiere de más depuración y, sobretodo, ajuste del algoritmo de filtrado acorde a las diversas semánticas que TCP aplica a sus segmentos.

6. Bibliografía

Bibliografía

1: Yonan, J, OpenVPN, 2002, http://openvpn.net/
2: Jabber Software Foundation, Jabber Overview, 2005,
http://www.jabber.org/about/overview.shtml
3: Gobierno Nacional Argentino, TELECOMUNICACIONES, 2004, http://tinyurl.com/c4g33
4: H. Kennedy, Binary Lexical Octet Ad-hoc Transport, 2002,
http://www.ietf.org/rfc/rfc3252.txt
5: Schneier, Bruce, , 1999, http://www.schneier.com/paper-pptpv2.html
6: IETF: Network Working Group, IP Security Document Roadmap, 1998,
http://www.ietf.org/rfc/rfc2411.txt
7: IETF: Network Working Group, Internet Security Association and Key Management
Protocol (ISAKMP), 1998, http://www.ietf.org/rfc/rfc2408.txt
8: D Harkins, D Carrel, The Internet Key Exchange (IKE), 1998,
http://www.ietf.org/rfc/rfc2409.txt
9: Wikipedia, IPv4 address exhaustion, 2004,
http://en.wikipedia.org/wiki/IPv4_address_exhausti
10: , IETF: Network Working Group, 2005, http://www.ietf.org/rfc/rfc3947.txt
11: , Free Software Foundation, , http://www.fsf.org
12: Lenhart, Madden, Hitlin, Teens and Technology, 2005,
http://www.pewinternet.org/pdfs/PIP_Teens_Tech_Jul
13: Jabber Software Foundation, Extensible Messaging and Presence Protocol (XMPP):
Core, 2004, http://www.ietf.org/rfc/rfc3920.txt
14: Jabber Software Foundation, Extensible Messaging and Presence Protocol (XMPP):
Instant Messaging and P, 2004, http://www.ietf.org/rfc/rfc3921.txt
15: Jabber Software Foundation, Jabber Enhancement Proposals, 2004,
http://www.jabber.org/jeps/
16: Musthaler, Linda, What's your company's policy on instant messaging?, 2005,
http://tinyurl.com/dxf5w
17: Guido Van Rossum, Python Programming Language, -, http://www.python.org/
18: Nezhdanov, A., Xmpppy: Jabber Python Library, 2005, http://xmpppy.sourceforge.net/

19: Titz, O., Why TCP Over TCP Is A Bad Idea, 2001, http://tinyurl.com/7ardy
20: Bellard, F., QEMU CPU Emulator, 2004, http://fabrice. bellard. free. fr/qemu
21: The Open Group, POSIX IEEE Std 1003.1, 2004 Edition - Sockets, 2004, http://www.opengroup.org/onlinepubs/009695399/func
22: POSIX, POSIX 1003.1e DRAFT 17, , http://www.guug.de/~winni/posix.1e/download.html
23: Allman, M., Paxson, V. and W. Stevens, TCP Congestion Control, 1999, http://www.ietf.org/rfc/rfc2581.txt
24: IETF, Requirements for Internet Hosts -- Communication Layers, 1989, http://www.ietf.org/rfc/rfc1122.txt
25: M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowledgement Options, 1996, http://www.ietf.org/rfc/rfc2018.txt
26: V. Jacobson, R. Braden, D. Borman, TCP Extensions for High Performance, 1996, http://www.ietf.org/rfc/rfc1323.txt
27: Paul E. Black, ed., NIST - Dictionary of Algorithms and Data Structures, 2005, http://nist.gov/dads/
28: Ciarlante, J., UDPv6 support for OpenVPN, 2004, http://tinyurl.com/9pwdb
29: Ciarlante,J, README.IPv6, 2004, http://tinyurl.com/964v8
30: , General Public License, , http://www.fsf.org/licensing/licenses/gpl.html

7. Apéndice A: Código fuente de OXG

Archivo **README**:

```
# $Id: README,v 1.6 2005/08/26 00:38:18 jjo Exp $

OvenVPN XMPP Gateway  (sort of openvpn over jabber tunnel)

* Goal:
  To allow tunnelling openvpn through "normal" (ie. <message>) jabber
streams, thus
  avoid requiring (at least one side) public PoP.

* Requirements
  - OpenVPN knowhow
  - python (std installation, I've used 2.3)
  - xmpppy: download from xmpppy.sourceforge.net (or check if your
distro provides it)
    Python module "xmpp" must be found by python, if not installed by
system, the easiest
    way could be:
    a) use tar.gz file
       cd /path/to/openvpn-prox-jabber-py
       tar zxvf /path/to/download/xmpppy-0.x.y.tar.gz
       ln -s xmpppy-0.x.y/xmpp .
    or
    b) use CVS
       cd /path/to/openvpn-prox-jabber-py
       cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/xmpppy
co -d xmpppy-cvs -P xmpppy
       ln -s xmpppy-cvs/xmpp .

* Installation
  None yet, run the script from the directory itself, eg:
     ./main.py <conf_file>

  To create a conf_file, do run "./main.py" alone, it will output a
conf_file example:
     ./main.py > oxg.home.conf
     vim oxg.home.conf
     ./main.py oxg.home.conf

* Usage examples
  1) You want to jabbertunnel UDP/1194 between A <-> B, ie:

     UDP/1194<-->[ A ]<=======jabber==========>[ B ]<-->UDP/1194
                    |                            |
```

```
        user@jabber.org/vpnA            user@jabber.org/vpnB

     You must run:
       user@A$ ./main.py oxg.AtoB.conf &
       root@A# openvpn --remote localhost --rport 1194 --lport 4999 ...

       user@B$ ./main.py oxg.BtoA.conf &
       root@B# openvpn --remote localhost --rport 1194 --lport 4999 ...

     Actually lport must be anything except rport because it's already
bound by
     this gateway.
```

Archivo **main.py:**

```
#!/usr/bin/python
# $Id: main.py,v 1.30 2005/07/26 03:28:32 jjo Exp $
# vim: sw=2:ts=2

#
# OXG: OpenVPN XMPP Gateway: encapsulate UDP datagrams via Jabber
clients (XMPP streams)
#
# Should investigate and use IBB, xmpppy does (experimentaly) implement
it
#
import os
import sys
import socket
import asyncore
import errno

from parseconf import *

import msgudp
import msgjab
import msgfile
import encoding

# __main__
try:
     conf=parseargs(sys.argv)
except:
     sys.exit(1)

buffer_size=8192

def main():
     udp=msgudp.msgudp()
     jab=msgjab.msgjab(conf.my_jid, conf.password, conf.peer_jid,
debug=conf.debug)
     stdinmsg=msgfile.msgfile(sys.stdin)
```

```
     udp.bindport(conf.udpport)
     udp.set_msg_target(jab, encoding.tobinary, encoding.toascii)
     jab.set_msg_target(udp)
     stdinmsg.set_msg_target(jab)

     jab.connect(conf.server)
     jab.sendinitpresence()
     for x in [udp, jab, stdinmsg]:
          x.async_init()
     asyncore.loop()

main()
```

Archivo **msgio.py:**

```
# $Id: msgio.py,v 1.4 2005/08/17 03:47:30 jjo Exp $
#
# vim: sw=2:ts=2
import asyncore
class msgio:
     msg_target=None
     msg_encode=None
     msg_decode=None
     def msg_send(self,m): raise NotImplementedError()
     def msg_recv(self): raise NotImplementedError()
     def msg_recv_loop(self):
          while True:
               m=self.msg_recv()
               if (m==None):
                         break
               try:
                         if (self.msg_decode): m=self.msg_decode(m)
                         if (m==None): continue
                         print "m=", m
                         if (self.msg_target.msg_encode):
m=self.msg_target.msg_encode(m)
                         if (m==None): continue
                         self.msg_target.msg_send(m)
               except:
                         continue
     def set_msg_target(self, msg_target, msg_encode=None,
msg_decode=None):
          self.msg_target=msg_target
          self.msg_encode=msg_encode
          self.msg_decode=msg_decode

class asyncmsgio(msgio,asyncore.dispatcher):
     def writable(self): pass
     def handle_connect(self): pass
     def handle_read(self):
          self.msg_recv_ready()
```

```
          self.msg_recv_loop()
     def handle_error(self): raise "msgio.handle_error() exception"
     def async_init(self):
          asyncore.dispatcher.__init__(self, sock=self.getsock())
```

Archivo **msgudp.py:**

```
# $Id: msgudp.py,v 1.7 2005/07/26 03:28:33 jjo Exp $
# vim: sw=2:ts=2
import os
import sys
import socket
import msgio


class msgudp(socket.socket, msgio.asyncmsgio):
     def __init__(self):
          socket.socket.__init__(self,socket.AF_INET, socket.SOCK_DGRAM)
          self.__queue=[]
          self.last_src=()
     def bindport(self,port):
          self.port=int(port)
          if self.port:
               bindaddr=('', self.port)
          self.bind(bindaddr)

     def __enqueue(self, e):
          return self.__queue.append(e)
     def __dequeue(self):
          return self.__queue.pop()

     def getsock(self):
          return self
     def msg_recv_ready(self):
          m,self.last_src=self.recvfrom(8192)
          print "msgudp.recvfrom len=%d <-" % len(m), self.last_src
          self.__enqueue(m)
     def msg_send(self,m):
          if (self.last_src):
               print "msgudp.msg_send ->", self.last_src
               try:
                         self.sendto(m, self.last_src)
               except:
                         self.last_src=()

     def msg_recv(self):
          m=None
          try:
               m=self.__dequeue()
          finally:
               return m
```

Archivo **msgjab.py:**

```
# $Id: msgjab.py,v 1.17 2005/08/17 03:47:30 jjo Exp $
# vim: sw=2:ts=2
import xmpp
import re
import msgio
class msgjab(xmpp.Client,msgio.asyncmsgio):
     def __init__(self, jid, passwd, peer_jid, debug=0):

self.JID_RE=re.compile(r'(?P<user>^[\w.]+)@(?P<domain>[\w.]+)(/(?P<resou
rce>[\w.-]+))?')
          jid_dict=self.__jid_match(jid)
          self.user=jid_dict.get("user")
          self.domain=jid_dict.get("domain")
          self.resource=jid_dict.get("resource")
          self.host=self.domain
          self.debug=debug

          self.peer_jid=peer_jid
          self._passwd=passwd
          self.__queue=[]
          self.create_client()

     def create_client(self):
          #self.messageHandler=None
          self.xmpp_client=xmpp.Client(self.domain, debug=self.debug)
          self.xmpp_client.msgjab=self

     def connect(self,host=None,port=5222):
          if host:
               self.host=host

          if not self.xmpp_client.connect(server=(self.host,port)):
               raise IOError('Can not connect with jabber server "%s".'
% self.host)
          if not
self.xmpp_client.auth(self.user,self._passwd,self.resource):
               raise IOError('Can not auth with jabber server "%s".' %
self.host)

self.xmpp_client.RegisterHandler('message',msgjab_messageHandler)
     def disconnect(self):
          self.xmpp_client.disconnect()
     def _enqueue(self, e):
          return self.__queue.append(e)
     def __dequeue(self):
          return self.__queue.pop()
     def __jid_match(self, jid):
          return self.JID_RE.match(jid).groupdict()
```

```
     def sendinitpresence(self):
          self.xmpp_client.sendInitPresence(0);
     def fileno(self):
          return self.xmpp_client.TCPsocket._sock.fileno()
     def getsock(self):
          return self.xmpp_client.TCPsocket._sock
     def msg_send(self, msg):
          print "msgjab.msg_send ->", self.peer_jid
          return self.xmpp_client.send(xmpp.Message(self.peer_jid,msg))
     def msg_recv_ready(self, timeout=None):
          self.xmpp_client.Process(timeout)
     def msg_recv(self):
          m=None
          try:
               m=self.__dequeue()
               print "msgjab.msg_recv <-", self.peer_jid
          finally:
               return m

def msgjab_messageHandler(conn, mess_node):
          "messageHandler callback for xmpppy, must be static"
          conn._owner.msgjab._enqueue(mess_node.getBody())
          #messageHandler=staticmethod(messageHandler)
```

Archivo **msgfile.py:**

```
# $Id: msgfile.py,v 1.1 2005/07/03 16:06:03 jjo Exp $
#
import msgio
import asyncore
class msgfile(msgio.msgio,asyncore.file_dispatcher):
     def __init__(self,file):
          self.file=file
     def async_init(self):
          asyncore.file_dispatcher.__init__(self, self.file.fileno())
     def writable(self): pass
     def handle_connect(self): pass
     def handle_read(self):
          self.msg_target.msg_send(self.file.read())
```

Archivo **encoding.py:**

```
# vim: sw=2:ts=2
import base64

def toascii(m):
     return "base64:" + base64.encodestring(m)

def tobinary(m):
```

```
     if (m[:7]!="base64:"):
          print m
          return None
     m=base64.decodestring(m[7:])
     return m
```

Archivo **parseconf.py:**

```
# $Id: parseconf.py,v 1.3 2005/08/17 03:47:30 jjo Exp $
class parseconf(object):
     vars_req= ['my_jid', 'peer_jid', 'password', 'udpport']
     vars_opt= {'server':None, 'debug':0, 'visible':0}
     def __init__(self, conf_file):
          # parse conf_file, create dictionary w/found  varname = val
          conf_params={}
          for line in open(conf_file).readlines():
               try: key,val=line.strip().split('=',1)
               except: continue
               if (key[0]=='#'): continue
               conf_params[key.lower()]=val

          # assign required vars as config object attributes
          for varname in self.vars_req:
               try:
                         self.__setattr__(varname,
conf_params[varname])
               except:
                         print "\nERROR: missing '%s' variable in %s
config file\n" % (varname, conf_file)
                         sys.exit(1)
          # assign optional vars as config object attributes (w/default)
          for varname, val in self.vars_opt.items():
               try:
                         self.__setattr__(varname,
conf_params[varname])
               except:
                         self.__setattr__(varname, val)

def parseargs(argv):
     try:
          conf_file=argv[1]
     except:
          print """
usage: %s <conf_file>

     conf_file example:
     #required variables: %s
     my_jid=user@some.jabber.org/v1
     peer_jid=user@some.jabber.org/v2
     password=s0mep4SS
     udpport=1194
     #optional variables: %s
```

```
    server=actual.server.address.org

         """ % (argv[0], parseconf.vars_req, parseconf.vars_opt.keys())
         raise Error, 'missing conf_file'

    return parseconf(conf_file)
```