kleiner kommunikation
max@kleiner.com

1 OpenVPN mit Delphi

Eine häufig genutzte VPN-Software ist OpenVPN. OpenVPN setzt nicht auf neue, eigene Protokolle und Schlüsselaustauschverfahren auf, sondern nutzt mit dem TLS/SSL-Protokoll das bekannte OpenSSL (im Tunnel) als freie Implementierung. Licht in den Tunnel bringt folgender Bericht.

1.1 Durch den Tunnel

Ein VPN (Virtual Privat Network) ist eine Lösung um entfernte Netze via Tunnel untereinander gesichert zu verbinden. Es verwendet wahlweise UDP oder TCP als Protokoll und benutzt zur Verschlüsselung und Authentisierung die Bibliotheken von OpenSSL.

Nun was ist ein Tunnel? Meistens kommuniziert man nicht direkt mit einem Host, sondern über einen Proxy (Stellvertreter). Wenn also eine Anwendung eine Socket-Verbindung durch einen Proxy öffnet, ist Tunneling im Einsatz (spezielle Verfahren wie SOCKS5[1] seien außer Betracht).

Das schwierige beim Programmieren sind die asynchronen Sockets, die das Einrichten eigener Behandlungsroutinen mit dem Proxy erfordern, sofern keine Komponenten verfügbar sind.

Bevor ich in den Code eintauche, sei das „offizielle“ und kompilierte OpenVPN [1] erwähnt. Die aktuelle Version basiert auf der 2.0.9 und wurde ursprünglich als Mitbewerber zu IPSec entwickelt. Wie Bruce Schneier und Niels Ferguson meinen: “IPSec is too complex to be secure” [SF99]. Die Version setzt eine OpenSSL Installation voraus und wird (Abb. 2) durch einen Assistenten begleitet. Nach dem Konfigurieren, Einrichten des Routing und Starten des jeweiligen Skripts legt OpenVPN sowohl auf der Client- als auch auf der Serverseite eine virtuelle Netzwerkschnittstelle an (wenn nicht schon z.B. durch VMWare vorhanden), welche jeweils ein Ende des Tunnels definiert.

Der gesamte Datenverkehr läßt sich dann durch den Tunnel mittels SSL (siehe Heft 2/2009) verschlüsseln. SSL-Chiffrierung wird heute vor allem mit https eingesetzt, jedoch ist auch eine host to host oder Gateway -Verbindung im TCP-Kanal möglich. Vom Prinzip her (siehe Abb. 1) benötigt man keine PKI mit Zertifikaten (dies erstaunt), es genügen sogenannte „pre-shared keys“ basierend auf dem symmetrischen Verschlüsselungsverfahren, indem man den Schlüssel vorgängig den Partnern verteilt. Für eine Punkt zu Punkt Verbindung ist dieses Verfahren robust und effizient zugleich, sofern man als Einmaldefinition den Schlüssel über einen sicheren Kanal transportiert.

Wenn wir schon im Tunnel stecken, seien zwei weitere Verfahren erwähnt. Mit entweder „File transfer per SSH“ (SFTP) oder „FTP over SSL/TLS“ (FTPS) sind zwei komplett unterschiedliche, inkompatible Varianten bekannt! SFTP ist ein eigenes Protokoll und hat eigentlich mit FTP nicht direkt etwas zu tun. FTPS wiederum ist eine Erweiterung zu FTP, welche sämtliche TCP-Verbindungen per SSL verschlüsselt, jedoch FTP als Protokoll voll und ganz unterstützt.

Nebenbei: Mit den Indy Sockets und mir bekannten Komponenten bspw. funktioniert nur FTPS, SSH ist hingegen bei Administratoren und der Linux–Welt äußert beliebt.

[1] Das **SOCKS-Protokoll** ist ein Internet-Proxy-Protokoll

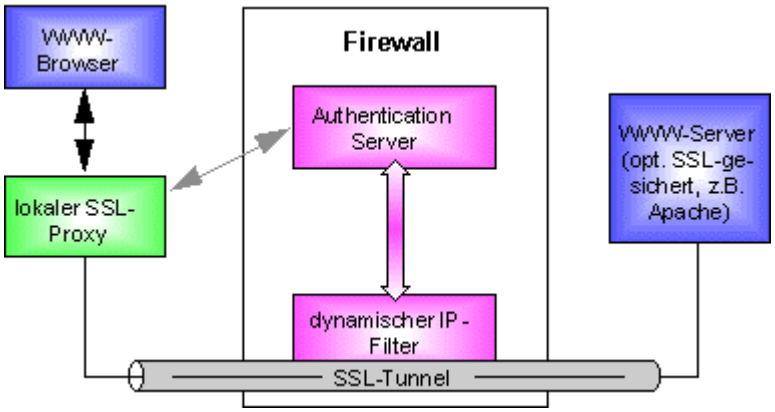


Abb. 1: Prinzip eines Tunnels

//ssl_tunnel_prinzip.gif [TU Dresden]

Bei der konkreten Implementierung hat OpenVPN „sicher“ einen klaren Vorteil gegenüber IPSec und den erwähnten Varianten. Es ist wesentlich leichtfüßiger, weniger komplex, läuft komplett im Userspace (d.h. Ring 3)[2] und kann seine Root-Rechte unmittelbar nach dem Start wieder abgeben. In einer Laufzeitumgebung begnügt sich OpenVPN zudem während der gesamten restlichen Zeit als unprivilegierter Prozeß. Hier muß mal gesagt sein, daß IPSec im Ring 0 eines Betriebssystems, die Situation auch nicht immer stabil halten kann.

Dazu kommen noch andere, nicht direkt sicherheitsrelevante Aspekte wie NAT-Fähigkeit, Proxy-Dienste, Firewall Durchgänge und ein unkomplizierter Umgang mit dynamischen IP-Adressen, selbst in Situationen, wo beide Endpunkte dynamische Adressen haben. Demgegenüber sind die Protokolle von IPSec standardisierter, doch OpenVPN hat mittlerweile von der IANA eine eigene Portnummer bekommen (UDP 1194), was für mich so aussieht, als wäre es auf dem besten Weg standardisiert zu werden.

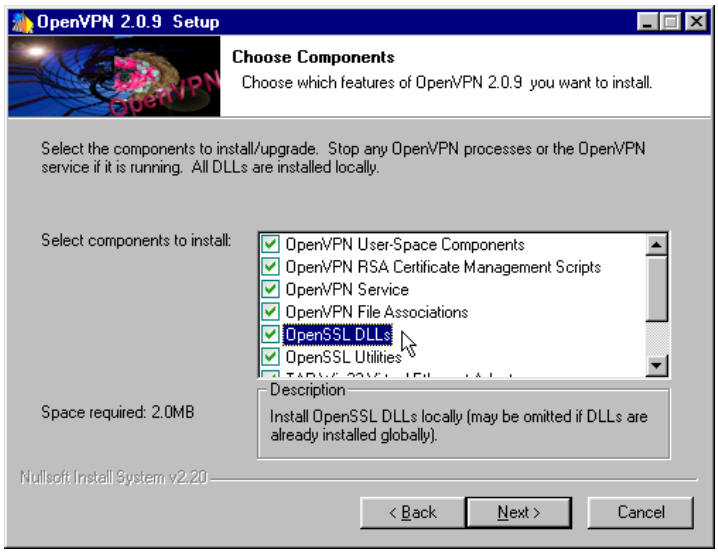


Abb. 2: In wenigen Minuten zum ersten Tunnelbau

1.2 Anschauungsunterricht

Das nun folgende Szenario mit Delphi und den beiden Beispielen Tunnel / SSL Tunnel hat außer den SSL-Bibliotheken keinen direkten Zusammenhang mit der OpenVPN Installation, es dient der

[2] Ein OS hat Ring 0 (Kernel), 1 (Dienste) und 3 (Apps)

Verständlichkeit wie ein konfigurierter Tunnel mit Hilfe eines Proxy funktioniert (oder eben nicht). Die in Delphi vorhandenen Komponenten stammen wiederum aus den Indy Sockets. Alternativ gibt's auch die SecureBlackBox, die kostenpflichtig ist.

http://www.eldos.com/sbbdev/desc-ssl.php

Für die Arbeit mit SSL benötigt man ja grundlegend die zwei DLL's (*ssleay32.dll* und *libeay32.dll*). In der Tunnel Demo (siehe Abb. 4) sind die beiden Komponenten (Abb. 3) Master und Slave enthalten. Die Unit `TIdTunnelMaster` ist ein Nachfahr von `TIdTCPServer`. Als Controller eines IP-Tunnels reagiert sie als Proxy für Anwenderverbindungen und ermöglicht eben durch Tunneling den Aufbau chiffrierter Verbindungen. Innerhalb des Tunnels entsteht ein Teil des privaten Netzes. D.h. die lokalen Daten lassen sich an eine externe IP-Adresse der Maschine übertragen, die an die gewählten Ports weiterleitet. Man spricht auch von lokalem „Port-Forwarding".

Der große Vorteil eines Proxy ist auch daß ich zum Proxy-Server im Prinzip eine HTTP-Verbindung herstellen kann, um sich dann alle möglichen Binärdaten/Textdateien mit jedem Protokoll auf jedem Port durchschleusen lassen, wenn dynamisches Port-Forwarding möglich ist!

Nun bauen wir den Tunnel auf:

Im Quelltext benötigt man vom Master und Slave die Hostadresse und Portnummer. Die Verwendung von localhost mit 127.0.0.1 ist eine Vereinfachung. Im Test brauchen wir den http-Server mit Port 80 und den firefox Browser, der als http Proxy mit Port 8080 einzustellen ist (im firefox unter Optionen); der zeigt dann auf den Slave:

```
Slave.DefaultPort:= 8080; //http proxy im Browser
```

Die *demotunnel.exe* wiederum kommuniziert im Port 9000 sozusagen als Brücke zwischen Client (Slave) und Server (Master). Jeder Aufruf durch den Browser erzeugt ein kurzes Aufzählen im Slave Part des Clients (Abb. 4) und ein sogenanntes `GET /safebrowsing/update` beim Server. Von der Interpretation ist folgendes festzuhalten:

Im Master Part bedeutet Slave die Exe selbst und Service ist der http-Server gemeint. Im Slave Part reagiert firefox als Client. Man benötigt also für jeden Port, den man tunneln will einen Slave, der mit dem Master in Verbindung steht und weiterleitet. Bspw. kann man Port 4512 über Port 80 laufen lassen indem der Master als Exe auf einem anderen Rechner liegt und fertig ist der Tunnel.

```
Master:= TIdTunnelMaster.Create(self);
  Master.MappedHost:= '127.0.0.1'; //ohne http://...eingeben!
  Master.MappedPort:= 80;
  Master.LockDestinationHost:= True;
  Master.LockDestinationPort:= True;
  Master.DefaultPort:= 9000;
  Master.Bindings.Add;
  Slave:= TIdTunnelSlave.Create(self);
  Slave.MasterHost:= '127.0.0.1'; //Proxy IP
  Slave.MasterPort:= 9000; //bridge zu master
  Slave.Socks4:= False; //oder Socks5
  Slave.DefaultPort:= 8080; //http proxy im Browser
  Slave.Bindings.Add;
.......
```

Die Anwendung erzeugt zur Laufzeit Fehler. Dies in Abhängigkeit vom Netz und Betriebssystem. Mit einem Proxy kann man ja keine direkten Verbindungen aufbauen. Ein Proxy sendet als Folge einer Anfrage vom Client eine weitere Anfrage an das eigentliche Ziel und leitet die Antwort des Zielservers an den Client zurück und dort treten dann Probleme auf. Es gibt also 2 Verbindungen im Netz:

Client<->TunnelSlaveServer(Proxy) + TunnelMasterServer(Proxy)<->Zielserver(Service).

Aus der Indy Hilfe [2] ist ersichtlich:

„When a TIdTunnelSlave becomes active, it initializes the Host and Port properties for the server's internal TCP connection that will acts as the encapsulated tunnel link to the TIdTunnelMaster server." Eine Abhilfe des Problems ist leider situativ und von der Konfiguration abhängig, z.B. kann man auch Verbindungen über andere Ports umleiten, indem Master und Slave auf dem gleichen PC sind.

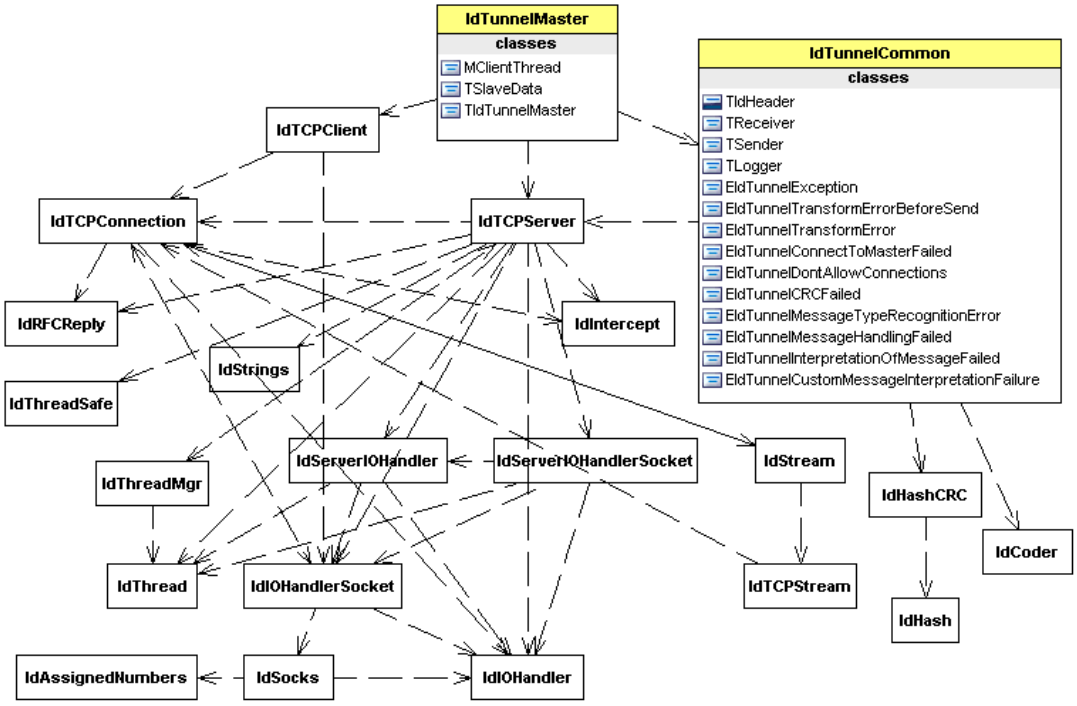


Abb. 3: Namensraum der Unit IdTunnelMaster

Als zweites Beispiel sei der SSL Tunnel erwähnt, der leider nicht vollständig funktioniert. Für diesen Tunnel sind drei Komponenten im Einsatz (oder Dreisatz): `TidTCPClient`, `TidMappedPortTCP` und `TidConnectionInterceptOpenSSL`. Unter Indy 10 gibt's die Tunnelkomponenten nicht mehr und wir sind mit dem Umbau noch nicht fertig.

Wirklich neu ist die Unit *TidMappedPortTCP.pas*. die wir kurz anleuchten. Die in der Unit enthaltene Komponente `TidMappedPortTCP` implementiert auch einen Proxy Dienst und ist ein Nachfahr der Klasse `TIdTCPServer`.

Die Dienste Telnet und Pop3 sind als eigene Klassen in der Unit vorhanden. Mit der Unit läßt sich so nebenbei auch ein einfacher Port-Scanner bauen. Verlassen wir mal den Browser und wechseln zu InterBase. Ein Szenario für einen chiffrierten Datenkanal von InterBase kann so aussehen:

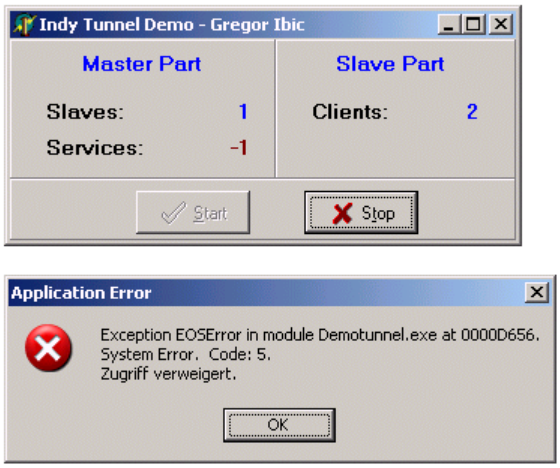
```
mPT:= TIdMappedPortTCP.Create(self); //mPT is TIdMappedPortTCP
mPT.MappedHost:= '192.168.52.31'; // interbase db server
mPT.MappedPort:= 3050; //interbase
mPT.DefaultPort:= 3051;
mPT.Active:= true;
```

Der Aufruf erfolgt sogleich mit einem Connection String als `127.0.0.1/3051` zum Proxy. Man achte, daß der im Code gesetzte default Port noch frei ist, als lokale Testumgebung kann auch `Defaultport:= 8080; MappedPort:= 80;`

mit einem Apache dienen. Die Konfiguration mit SSL ist dann ziemlich happig, return Werte und Streams von redirections sind falsch, so daß wir mittlerweile auf Synapse mit SSH ausgewichen sind.

http://synapse.ararat.cz/doku.php/

Dazu ist auch ein Testdemo vorhanden.



//indy_tunnel_app.tif

Abb. 4: Tunnel im Zugriff

1.3 Die Schlüssel zur Tunnelöffnung

Mit OpenVPN kann man die Authentifizierung auf zwei verschiedene Arten regeln: mit Zertifikaten oder mit den erwähnten „pre-shared keys“. Bei diesen handelt es sich um einen gemeinsamen Schlüssel mit Passwort, der vor der Nutzung auf beiden VPN-Endpunkten eingerichtet oder verteilt wird und somit vorgängig austauschbar ist. Das erscheint recht einfach – schließlich braucht man keine X.509-PKI-Infrastruktur für die Zertifikate einzurichten, aber begrenzt doch die Möglichkeiten zwischen n-Partnern eine Vertrauensstellung aufzubauen.

Preshared-Keys eignen sich für kleine VPN-Netze, mit wenigen Teilnehmern, wenn es keine Rolle spielt, daß n Teilnehmer den gleichen Schlüssel besitzen.

Zertifikate sind die wesentlich sichere Methode zur Authentisierung. Ein Client muß ein gültiges Zertifikat vorweisen, um sich am VPN-Server anzumelden. Übrigens ist bei den „pre-shared keys“ die Authentisierung mit der Verschlüsselung verbunden. Wenn kein korrekter Schlüssel vorhanden ist, läßt sich keine verschlüsselte Verbindung aufbauen.

Abschließend ein Tip zur Netzanalyse. Wer den Protokollen von SSL im Tunnel nicht traut, kann mit dem Tool wireshark in den Tunnel reinschauen. Wireshark 1.0.6 (vormals Ethereal) ist ein freier Netzwerk-sniffer, der den Datenverkehr im Netz analysiert und protokolliert (Ja richtig, man kann auch Protokolle protokollieren). In vielen Fehlersituationen oder Unstimmigkeiten erleichtert Wireshark das Auffinden der Fehlerquelle und erkennt Standardports durch eigene Symbolik und Farbgebung.

http://www.wireshark.org/

Errare humanum est. - Irre sind auch nur Menschen.

Ich wünsche guten Frühlingsstart und bis zur nächsten Folge „OpenGL in Delphi“.

Max Kleiner

Sourcen auf der CD oder Site entwickler.de

- tunneldemo_delphi_source.zip
- synapse_ssh_test_sourcen_delphi.zip

http://www.softwareschule.ch/download/tunneldemo_delphi_source.zip

Links:

[SF99] Schneier. A Cryptographic Evaluation of IPSec:
http://www.schneier.com/paper-ipsec.pdf
[1] http://www.openvpn.org/
[2] Indy Knowledge Base: http://www.indyproject.org/KB/
http://www.indyproject.org/Sockets/Docs/index.DE.aspx