A Security Solution For Wireless IP Networks

EPFL Semester Project

Jean-Philippe Pellet
jean-philippe.pellet@epfl.ch

supervised by

Dr. Christine Vanoirbeek, christine.vanoirbeek@epfl.ch
EPFL, I&C School

Dr. Mohamed Mancona Kandé, mohamed.kande@condris.com
Condris Technologies Sàrl

Arnaud Burlet, arnaud.burlet@condris.com
Condris Technologies Sàrl

14th April, 2005

Contents

1 Introduction

The goal of this projet is to find a convenient method to secure a wireless network. Security in this context means protecting the data transferred from being accessible to unauthorized third party, and preventing unauthorized third party from using the network to transfer data. For a more detailed requirements analysis, see Section 3.

Wired Equivalent Privacy (WEP) is currently the most popular security mechanism for wireless networks conforming to the IEEE 802.11a/b/g protocols. In Section 2.1, we shortly discuss why WEP is not secure and why an alternative to it must be found.

In this report, we propose two solutions to overcome the limitations of WEP.

- a solution based on Wi-Fi Protected Access (WPA), a set of protocols specifically designed for wireless networks;
- a solution based on Virtual Private Network (VPN) software.

The latter is chosen for this project because of the difficulties found in the deployment of a WPA solution as a semester project. The realization of the VPN solution is detailed in this report in Sections 3 and 4. Different VPN solutions are examined in Section 3.1: IPSec, PPTP, vTun and OpenVPN. OpenVPN is the resulting software from an open source cross-platform project; is is chosen for its ease of use and its cross-platform capabilities as well as its open source status.

OpenVPN is a command-line tool, which is not adequate for all users of the network type we aim at securing. A simple GUI wrapper for the command-line tool has thus been designed and implemented, which is detailed in Section 5.

Finally, the problem of automatic on-the-fly configuration file generation is discussed in a more general context than OpenVPN's only. An implementation of a tool using the Velocity library is presented in Section 6.

2 Defining The Problem

Wireless networks have been more and more used over the past few years. Once set up, they offer a fast and convenient way for several computers or mobile devices to exchange data. Although its general transfer speed cannot yet be compared with that of today's wired networks, the offered data rate has been growing to offer a theoretical maximum of 54 Mbps in standards such as IEEE 802.11a or 802.11g. Further standards such as IEEE 802.16 (a.k.a. WiMAX) plan to overtake good old Ethernet's 100 Mbps in the near future.

An ambient wireless network frees its users from the need to draw cables across rooms. But, because of its very structure (as explained below), it has some security flaws compared to a traditional wired network:

- First of all, one cannot precisely delimit its range. Wireless networks complying to IEEE 802.11g for instance use electromagnetic waves with 2.4 GHz frequency. There is no simple way to block them, i.e. prevent aves from crossing the boundaries of an office or a building in which the wireless network is installed. Indeed, anyone can listen easily to what is being sent by the network members (which is called "eavesdropping"). This could not be done so easily with a wired network—one should first gain access to the wiring.
- If anybody can listen to the network, anybody can try and send packets through it too, if there are no security measures taken against it. There is no straightforward way to consolidate the network paths by checking where data transfer goes through and how to protect the network.

Although wireless networks are easy targets for attackers, network engineers did not provide users from the beginning with adequate security measures. Today, a lot of home wireless networks are unprotected and the communications unencrypted by default. This enables people such as neighbours to get a free internet access. The first implemented wireless security protocol designed for networks of type 802.11 is known as WEP—Wired Equivalent Privacy.

2.1 Wired Equivalent Privacy (WEP)

WEP is part of the 802.11 standard. It uses the RC4 algorithm for encryption and CRC-32 checksum for integrity preservation.

What makes WEP vulnerable is the fact that it ignores any key management protocol and relies on a shared key mechanism. The shared key is the same for all users of the wireless network.

In 2001, Fluhrer et al. demonstrated a way to find out the RC4 key after a few hours of eavesdropping a WEP-protected wireless network [1]. Freely available software soon automated the tracking process, thus making WEP very easily breakable.

"If an unprotected wireless network is like an open house, using WEP is similar to closing the door without locking it. The door still remains very easy to open." [2]

2.2 Wi-Fi Protected Access (WPA)

Wi-Fi Protected Access is an alternative to WEP that has been developed in order to address WEP's shortcomings, while remaining compatible with existing hardware. WPA seems to be successful at addressing most of these shortcoming. However, there are many adapter

cards for which no WPA driver is available. If that might not be a problem in a company, where everyone could be requested to use compatible hardware, it is surely a problem when dealing with public networks, for instance at airports or in hotels. Moreover, a complete WPA solution with full access control features has not been implemented yet, at least in a way that it is useable, customizable and/or open source. However, a part of the solution has been implemented as open source in FreeRADIUS [3].

It is beyond the scope of this project to discuss to implementation of WPA as a whole. Even implementing only parts of it would have required a very good understanding of WPA as a whole. Without any available reference implementation and because of the lack of appropriate hardware, implementing would be difficult to test and would have led only to a partially useable solution (if at all).

2.3 Virtual Private Network (VPN)

The next idea to emerge when discussing how to realize the security goal was to check what kind of protocols or infrastructures are used when security is needed on a untrusted network, be it wired or wireless. The solution of choice is Virtual Private Network (VPN).

VPN is not a standardized protocol or a well-defined set of protocols, but rather a generic denomination. A lot of VPN implementations exist, some of them using existing protocols such as IPSec or Microsoft's Point-to-Point Tunneling Protocol (PPTP) as main underlying security measure, some others using other methods to establish secure and encrypted connections between two participants. They are thus not all equivalent, and not all compatible with each other. Moreover, they will not all work with all three major platforms (Linux, Mac OS X and Windows 2000/XP) that we decided to support.

Anyway, their principles remain the same. A VPN software can build secure connections between two end points by encrypting sent data. Other data streams can then be tunneled through this existing secure connection.

A typical example is the following: imagine a company based in Lausanne and selling its products in Lausanne and Geneva. Both Geneva (G) and Lausanne (L) offices will want to use a common intranet in order to share files and transfer data. The internet between them cannot be trusted and the administrators are looking for an easy and secure way to connect both offices. Here is a general solution, using a VPN:

1. Choose two different local subnets for the computers in G and L;
2. Pick a machine named Gi in G and another machine Li in L which are both connected to the internet and build a secure tunnel connection between them using VPN software;
3. In G, redirect all traffic with a destination address belonging to L's subnet to Gi, which will encrypt it and send it securely to Li over the internet. Li will then decrypt the sent data and forward it on its own private network;
4. Do the same in L with traffic with a destination address belonging to G's subnet.

Similarly, point-to-point VPN connections could thus be used between wireless clients and an access point or gateway. Authentication features possibly integrated in the VPN software would allow some kind of access control, too.

The first part of the project is to use a VPN solution to secure authorized wireless connections and to deny internet access to unauthorized users.

3 Preparation of the Virtual Private Network

To facilitate the understanding of VPN, let us recapitulate what requirements are needed from the chosen particular solution.

The solution we are looking for must:

1. Encrypt the created IP tunnels (which is in principle guaranteed by any VPN solution) using industrial-strength algorithms. (*Data Encryption*)
2. Be able to distinguish between authorized and unauthorized clients. (*Access Control*)
3. Be scalable and able to serve hundreds of clients without introducing an increasing latency or a bad responsiveness as the number of clients increases. (*Scalability*)
4. Be cross-platform. The VPN server will have to run on a Linux box; clients will have Linux, Windows and Mac OS X[1] as main operating systems. (*Compatibility*)
5. Have a graphical, user-friendly interface for clients (this requirement does not pertain to the server software). (*Interface*)
6. Be royalty-free. Open source software is of course preferable. (*Licensing*)

If Requirement 1 (Data Encryption) comes with every VPN solution, that is not true for all other requirements. For instance, Requirements 2 and 3 (Access Control & Scalability) imply several more important requirement about the algorithms used. In particular, they forbid the use of a static key or pre-shared key algorithm and require the use of more sophisticated authentication methods like the ones used by the Transport Layer Security (TLS) protocol. TLS allows dynamic session key generation based on public/private key pairs.

Finally, Requirement 4 (Compatibility) requires the software to either exist on all considered platforms, or to be compatible with another piece of software that runs on platforms it does not directly support.

3.1 The VPN Software

The following potential solutions have been examined:

- **IPSec**. IPSec (short for Secure Internet Procotol) is an Internet Engineering Task Force (IETF) standard that provides encryption capabilities to IP. It is very flexible and proprietary extensions can be made to it. That is why there are many implementations available which should be compatible with each other, but in fact are often not, in particular with what pertains to authentication methods.
- **Point-to-Point Tunneling Protocol (PPTP)**. PPTP is a Microsoft VPN/tunneling protocol which includes an authentication scheme. User data is encrypted and encapsulated in traditionnal IP packets. PPTP has had security flaws in the past and is not considered today to be the most robust VPN solution [4].
- **VTun**. VTun is an open source VPN project that runs on a range of Unix flavors and uses a custom protocol. An analysis of this protocol showed some security flaws in the past [5].

[1] There were only very few Mac OS 9 portable computers with Wi-Fi capabilities; all of them are over five years old now. Choosing not to support Mac OS 9 does not represent a big customer loss, if at all.

- **OpenVPN**. OpenVPN is an open source project and includes a TLS-based VPN solution running on many platforms.

For the following reasons, OpenVPN seems to be a good candidate for almost all requirements described above: it uses OpenSSL[2] to encrypt data going through tunnels (see Requirement 1) and has a TLS mode to enable public key, certificate-based encryption rather than a pre-shared key mechanism (see Requirement 2). It runs on all target platforms as a single application (requiring the installation of an additional driver for Windows and Mac OS X) (see Requirement 5). When run in server mode, it allows many clients to build tunnels and serves them (see Requirement 3). It is open source and is distributed under the GNU Public Licence (see Requirement 6).

Requirement 5 alone (Interface) has yet to be fulfilled. As a command-line tool, OpenVPN is not adequate for use out of the box by all users. The final package will have to include additional GUI wrappers in order to hide the command-line commands and output behind a graphical interface familiar to all users.

3.2 Exploring OpenVPN

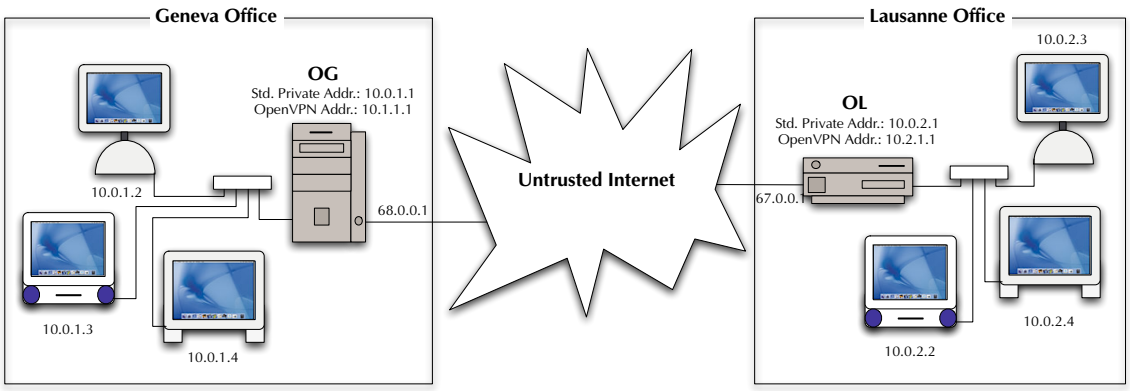
No matter what the run mode or encryption mode is, two OpenVPN instances trying to communicate will always first build a tunnel between them. By default, data to be transferred will be encapsulated in a new UDP package and sent to the other instance. (Either UDP or TCP can be selected and the port number to be used can be customized.) Encapsulation includes here encryption and possible packet splitting and reassembling. Data to be tunneled will be sent to a specific address, logically connected to the end point of OpenVPN's tunnel. This scenario corresponds to the situation described in Section 2.3.

Let us further demonstrate this example using two separate OpenVPN instances.

- The subnet in the Geneva office (G) is 10.0.1.0/24, the one in the Lausanne office (L) is 10.0.2.0/24.
- On a machine in G which has the standard private IP Address 10.0.1.1, an OpenVPN instance (OG) is running and listens for data to be transferred on the address, say, 10.1.1.1 (it must not be part of the private subnet). The public internet IP address of the machine is, say, 68.0.0.1.
- Similarly, a machine in L is also running OpenVPN (OL) with address 10.2.1.1 and has public IP address 67.0.0.1.

Now, if OG knows OL's public address, they can establish a tunnel. If they further know each other's private address, they can listen for data being sent to it, grab it from their respective private network, encrypt it, send it through the tunnel and have the other instance decrypt it.

[2]See Section3.3 for OpenSSL.



The next step is to add rules in order to route all traffic having to do with the respective subnets:

- In G, route all traffic with a destination in 10.0.2.0/24 through 10.2.1.1, OL's private address. OG will see it and tunnel it to OL.
- Similarly, route all traffic in L with a destination in 10.0.1.0/24 through 10.1.1.1.
- On the machines running OL and OG, enable forwarding so that incoming packets are forwarded to the whole respective subnets.

What is achieved here is a seamless secure interconnection of two private subnets over the untrusted internet using encrypted tunneling with OpenVPN.

3.2.1 Encryption Modes

In OpenVPN, there are three encryption modes:

- No encryption at all;
- Encryption based on a pre-shared key mechanism;
- Encryption based on a TLS-based dynamic key exchange mechanism using signed SSL certificates.

In pre-shared key mode, both partners must have agreed on a common set of four keys used for encryption, decryption and message integrity verification. These keys must be know prior to any reliable secure connection, because the secure connection uses them. Thus, they have to be transferred from one computer to the other somehow in a secure way, as both partners need to know all keys. That is why this mechanism is not easily scalable and can become cumbersome as soon as the number of client exceeds half a dozen. Dynamically creating and distributing new keys raises an issue, too.

The TLS-based mode will be chosen as the preferred method for the rest of this project because of its scalability and increased security compared to the pre-shared key mechanism. It allows both partners to communicate, encrypt and decrypt data without having to know in advance how the used keys will look like.

3.2.2 The Tranport Security Layer Protocol [6]

The TLS procotol was designed to run on reliable layers beneath application protocols such as HTTP or SMTP, or directly on top of TCP (just like its predecessor, SSL—Secure Socket Layer). OpenVPN, however, uses TLS over UDP: the developers introduced a reliable transport layer between UDP and TLS (note that it is only provided for TLS, not for post-TLS tunneled packets containing user data[3]). The main reason for choosing UDP and a single port rather than TCP is that it makes OpenVPN a lot more firewall-friendly and is thus less prone to be disturbed by too restrictive firewall blockings.

Here is a rough shape of a TLS start sequence[4]:

1. Negotiation between peers about what algorithms are to be used;
2. Exchange of public keys and certificate-based authentication;
3. Then, after authentication and subsequent key derivation, encryption (based on a symmetric cipher) is turned on.

OpenVPN uses the same symmetric cipher as WEP, namely RC4. What makes it more robust, is that the key is neither exposed during the opening negotiations nor later; it is derived by the secure TLS protocol using public keys and random data generated by both communication partners[5].

Why not use asymmetric ciphers to encrypt user data, and stick with RC4? Asymmetric encryption and decryption requires considerably more overhead than symmetric mechanisms and requires still today too much CPU time in order to be really deployed and used on end-user computers by multiple simultaneous streams sustaining an acceptable data rate without hogging the CPU.

In traditional uses of TLS, for instance in HTTPS, only the server has to authenticate. In OpenVPN however, both peers will authenticate to each other.

Up to now, the participating OpenVPN instances have been running both in the same mode, participating in a peer-to-peer connection. The introduction of TLS will designate one participant as the *TLS server*, the other one being the *TLS client*.

An OpenVPN instance running in TLS server mode can require the TLS clients' public keys to be signed by a so-called *certification authority*, from which it possesses the public key. It will then only accept incoming connections from users whose public key identity is certified by a trusted third party.

Alternatively (and more conveniently in the current case), OpenSSL can be used on the machine running the TLS server to generate a set of private and public keys which will be considered as the certification authority. New public/private key pairs can then be generated and signed using the newly-generated certification authority's private key on the fly, independently from a third party. The TLS server will also possess its own signed pair of keys for the mutual authentication.

3.2.3 The OpenVPN Protocol

The exact format of the tunnelled TCP or UDP packets will not be described here. It can be found at http://openvpn.net/security.

[3] If the tunneled session is TCP for instance, the TCP protocol itself will provide the reliability layer. This avoids the problem known as *reliability layer collision* described in [7].

[4] For a complete commented sequence diagram, see RFC [6] or [8].

[5] Key derivation details in[9].

3.3 OpenSSL [10]

OpenSSL is an open source project aiming at providing a robust implementation of SSL and TLS. It is used internally by OpenVPN; it can also be used in order to generate key pairs and to sign public keys using a certification authority's private key.

A few commands are particularly useful for on-the-fly key generation and certification; they are shortly presented here.

3.3.1 Root Certificate Generation

The following command generates a new certification authority certificate and private key.

`openssl req -nodes -new -x509 -keyout` *`ca.key`* `-out` *`ca.crt`*

ca.key will be the certification authority's private key and *ca.crt* its public key. The file *openssl.cnf* can then be edited in order to make the *certificate* variable point to *ca.crt*.

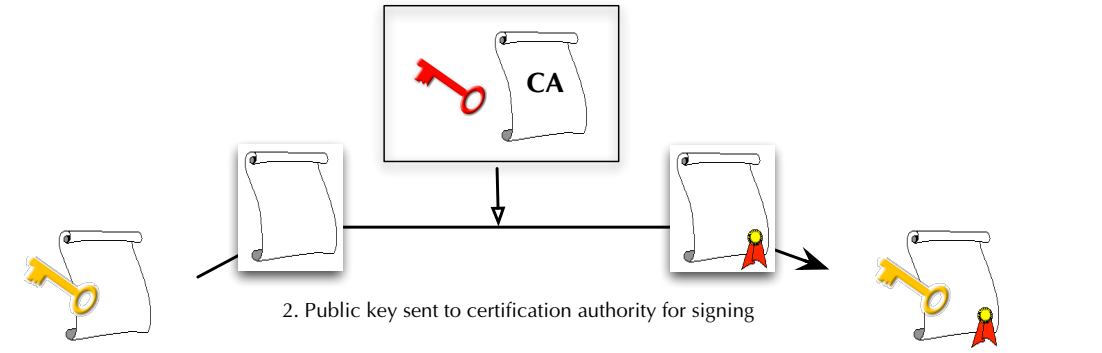
3.3.2 Signed Key Pair Generation

This operation is carried out in two steps. First, a private key and a temporary unsigned public key are issued, the latter being output as a *certification request*. This certification request file can be transported to the machine possessing the certification authority's private key in order to be signed (in the considered use case, it will stay on the same machine, because the machine generating key pairs is its own certification authority). The signed public key will be the required certificate. Note that the certification authority used to sign the newly-generated public key is the one designated by the *certificate* variable in the *openssl.cnf* file.

1. `openssl req -nodes -new -keyout` *`newkeypair.key`* `-out` *`newkeypair.csr`*
2. `openssl ca -policy policy_anything -out` *`newkeypair.crt`* `-in` *`newkeypair.csr`*

(Note that for certificate authority functions [`openssl ca` *`xxx`*], the file *index.txt* and *serial* must be set up. The former can be empty, the latter must be initialized to 01.[6])

The files *newkeypair.key* and *newkeypair.crt* are the new private key and public key, respectively. The file *newkeypair.csr* will not be needed any more and can be deleted.



[6] For more information, see [11].

3.3.3 Diffie-Hellman Parameter Generation

This operation generates a file containing Diffie-Hellman parameters. The Diffie-Hellman key agreement procotol is used by OpenVPN; the generated parameter file is needed by the TLS server. In short, Diffie-Hellman allows two partners wishing to use a symmetric encryption or signing mechanism to securely create a common key over an unstrusted communication medium without a third party being able to know it.[7]

`openssl dhparam -out` *`dh.pem`* `1024`

1024 is the length of the generated parameters in bits. *dh.pem* is the resulting parameter file.

[7] For more information, see [12].

4 Deploying OpenVPN

OpenVPN requires:

- A TUN/TAP driver. This allows it to control virtual point-to-point IP or Ethernet devices.
- The OpenSSL library. OpenSSL is required in order to use encryption.
- Optionally, the LZO real-time compression library. If compiled without it, OpenVPN will not be able to compress forwarded user data.
- Optionally on Linux and Solaris systems only, the Pthreads library. This library is used to reduce latency during TLS key negociations.

In this section, the availability of the requirements for the target platforms—Linux, Mac OS X and Windows 2000 and XP—will be inspected. Then, three simple use cases will be examined. Finally, OpenVPN's configuration files will be discussed, and the realistic use case of OpenVPN for our targeted wireless security will be described.

4.1 Platform Considerations

The OpenSSL library can be found at http://www.openssl.org/source/. The LZO library can be found at http://oberhumer.com/opensource/lzo/download/. Both should be installed before proceeding with OpenVPN[8].

4.1.1 Linux

The LZO library has to be installed. After downloading the OpenVPN source, a traditional `configure`, `make`, `make install` builds and installs OpenVPN. It is ready to be used.

4.1.2 Mac OS X

OpenSSL is preinstalled with Mac OS X. LZO has to be installed. Additionally, a TUN/TAP driver must be installed. The recommended driver can be found at http://www-user.rhrk.uni-kl.de/~nisler/tuntap/. `configure`, `make`, `make install` then builds and installs OpenVPN. *In order to use the TUN/TAP driver, OpenVPN must be run as root.*

4.1.3 Windows

Precompiled Windows binaries can be found at http://openvpn.net/download.html. Therefore, there is no need to download OpenSSL or LZO (as long as OpenSSL commands are not needed in order to generate key pairs or sign public keys). This installer conveniently installs a TAP driver. *Default installations of Windows XP and installations of Windows 2000 with Driver Signing Verification turned on will issue a warning prior to the installation of the TAP driver, indicating that it is not certified as being compliant to the Microsoft driver security policy.*

[8] For detailed compilation and installation instructions, see http://openvpn.net/install. Only the relevant options are mentionned here.

4.2 Three Test Cases[9]

The following simple test cases will demonstrate how to easily get OpenVPN running in different modes. For all examples, two machines are assumed to be up with IP adresses 192.168.0.1 and 192.168.0.2, respectively.

OpenVPN can be run by passing a configuration file as command-line parameter (see Section 4.3), or equivalently by specifying all options using as many command-line parameters. The examples here show the command-line parameters version.

4.2.1 Simple Unencrypted Tunnel

This example just aims at testing the creation of a tunnel. On 192.168.0.1:

```
openvpn --remote 192.168.0.2 --dev tun1 --ifconfig 10.0.0.1 10.0.0.2 --verb
6
```

On 192.168.0.2:

```
openvpn --remote 192.168.0.1 --dev tun1 --ifconfig 10.0.0.2 10.0.0.1 --verb
6
```

The *remote* parameter specifies the address of the communication partner. *dev* indicates which virtual device to use; multiple instance of OpenVPN running on the same machine cannot use the same virtual device. The two *ifconfig* parameters indicate the respective addresses of the tunnel's end points. *verb* specifies the verbosity and can be changed without affecting the configuration.

The tunnel can be checked using for instance on 192.168.0.1:

```
ping 10.0.0.2
```

10.0.0.2 represents the end point of the tunnel virtually located on the machine at address 192.168.0.2. OpenVPN at 192.168.0.1 listens for traffic with destination address 10.0.0.2 and tunnels it to 192.168.0.2. Similarly, the command

```
ping 10.0.0.1
```

issued on the machine at 192.168.0.2 tests the tunnel the other way round.

4.2.2 Static Key-Based Encrypted Tunnel

The tunnel will now be enhanced with encryption. Static key-based encryption works with pre-shared symmetric keys: a key must be generated somewhere and securely copied to the two communication partners. This key is the "shared secret" and must be kept safe as it is all that is needed to decrypt the communication.

OpenVPN can generate a static key with the following command:

```
openvpn --genkey --secret key
```

The file *key* can be generated on 192.168.0.1 and copied over to 192.168.0.2. Both partners can now start OpenVPN similarly, just adding the *secret* parameter. On 192.168.0.1:

```
openvpn --remote 192.168.0.2 --dev tun1 --ifconfig 10.0.0.1 10.0.0.2 --secret
key --verb 6
```

On 192.168.0.2:

```
openvpn --remote 192.168.0.1 --dev tun1 --ifconfig 10.0.0.2 10.0.0.1 --secret
key --verb 6
```

Pinging across the tunnel demonstrates that the connection and the tunnel set up were successful.

[9] Inspired by examples in the OpenVPN HOWTO.

4.2.3 TLS-Based Encrypted Tunnel

Static key encryption does not scale well for a large number of clients; all keys must be known in advance by both partners prior to any OpenVPN communication. Dynamic key generation using TLS is put into practice with the following example.

Previous sections have discussed TLS and its private/public keys and certification authority principles. One of the two partners has to be the TLS server, the other one being the TLS client. The server has its key pair, named *server.key* and *server.crt*, as well as the certification authority's certificate, *ca.crt* (which could have been generated by the server itself if needed). It also needs a Diffie-Hellman parameter file. The client also has its own key pair, *client.key* and *client.crt*, as well as *ca.crt* too. Both *server.crt* and *client.crt* must have been signed by the certification authority.[10]

192.168.0.1 is arbitrarily designated as the TLS server. OpenVPN is launched this way;

```
openvpn --remote 192.168.0.2 --dev tun1 --ifconfig 10.0.0.1 10.0.0.2 --tls-server
--dh dh.pem --ca ca.crt --cert server.crt --key server.key --reneg-sec 60
--verb 6
```

192.168.0.2 will run the TLS client:

```
openvpn --remote 192.168.0.1 --dev tun1 --ifconfig 10.0.0.2 10.0.0.1 --tls-client
--ca ca.crt --cert client.crt --key client.key --reneg-sec 60 --verb 6
```

Note the additional *dh* parameter for the server. The *reneg-sec* option specifies after how much time new keys will be generated. 60 seconds has been chosen for testing purposes only; a more realistic value would be 3600, thus causing an hourly key renewal rate.

As usual, pinging across the tunnel confirms that it is working.

4.3 Secure Wireless Networking Configuration[11]

The previous examples showed simple illustrations of how OpenVPN can easily be used to securely tunnel data streams. However, it is still far from representing the ideal solution. With the above examples, a new OpenVPN instance must be run for each new tunnel. A central server with n clients would thus have to have n copies of OpenVPN running, using n different tap/tun virtual devices. A more efficient and more scalable use has been devised and is explained below.

The configuration presented here now encompasses a config file for the server and a config file for clients. The server will now not only be a TLS server as before, but a true OpenVPN server, dealing with several clients at once and maintaining as many tunnels as there are clients. Moreover, this set up will enable IP traffic routing through the tunnel on the client side. (Please note that many server options are new OpenVPN 2 features and are not all fully documented yet.)

As mentioned before, OpenVPN can be run and instructed to read all options from a certain configuration file with this command:

```
openvpn --config config.ovpn
```

4.3.1 Client Configuration

The following content is saved in the file *main-client.ovpn*.

[10] Refer to 3.3 on page 11 in order to generate the required files.
[11] The parameters and run modes described here are only valid for OpenVPN 2.x, not OpenVPN 1.x.

```
client
dev tun
proto udp
remote 192.168.0.1 1194
resolv-retry infinite
nobind
persist-key
persist-tun
ca ca.crt
cert client.crt
key client.key
reneg-sec 3600
comp-lzo
verb 1
```

The *client* option designates the client mode. *proto* allows to choose between UDP and TCP. Several *remote* parameters can give the address and port number of OpenVPN servers (the default port being 1194). *resolv-retry infinite* tells OpenVPN to keep trying to reach a server. This is useful on laptops which are not always connected to a network, for instance. *persist-key* and *persist-tun* try to keep using the same resources upon subsequent restarts. *comp-lzo* enables LZO compression—only available if OpenVPN was compiled with the LZO library (if the tunneled stream is already compressed in some way, it can be useless to turn on this further LZO compression, which would only cause more overhead). The other options should be familiar.

4.3.2 Server Configuration

The following contents are saved in the file *main-server.ovpn*.

```
port 1194
proto udp
dev tun
ca ca.crt
cert server.crt
key server.key
dh dh.pem
server 10.0.5.0 255.255.255.0
ifconfig-pool-persist ipp.txt
push "redirect-gateway"
client-to-client
keepalive 10 120
comp-lzo
persist-key
persist-tun
status openvpn-status.log
verb 5
```

There are a few new options compared to the client configuration file. *port*, *proto*, *dev*, *ca*, *cert*, *key*, *dh, comp-lzo*, *persist-key* and *persist-tun* have already been discussed. The *server* option takes two parameters defining a subnet, from which clients will get their respective OpenVPN IP addresses. In this case, the server will always get the address 10.0.5.1, and the clients will get the addresses 10.0.5.2 and up. *client-to-client* prevents clients from "seeing" each other—they only see the server. *ifconfig-pool-persist* logs which client gets which address and tries to give the same address to clients reconnecting subsequently by storing these mappings in the file used as parameter. The *status* options logs OpenVPN events to the

file passed as parameter. *keepalive* takes two integer parameters. The first one indicates a rate in seconds at which clients will be ping'd; the second one tells after how many seconds of unreachability clients will be considered dead. Finally, the *push* option takes an other option surrounded by double quotes, which will be "pushed" to the client upon successful connection. In this case, clients are instructed to redirect IP traffic through the newly-created tunnel. Note that an additional corresponding line *redirect-gateway* in the client configuration file would have the same effect.

A few additional commands are necessary on the server side to ensure correct IP packet forwarding, so that they will not stay at the tunnel endpoint but continue their way to the internet[12]:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -A FORWARD -i tun+ -j ACCEPT
iptables -t nat -A POSTROUTING -j MASQUERADE
```

[12]For info about this use of *echo*, see http://www.redhat.com/docs/manuals/haserver/RHHAS-1.0-Manual/x4011.html; for info about *iptables*, see http://www.linuxguruz.com/iptables/howto/iptables-HOWTO.html.

5 GUI Connection Assistant

The application to be developed must present a graphical interface to OpenVPN's command-line use. It must allow users to connect and disconnect from the VPN, redirecting IP traffic to go through the tunnel once established. It must inform the user about common tunnel establishment failures, hinting at what the likeliest problem could be. Of course, it must be platform-independant, or at least run on Linux, Mac OS X and Windows 2000/XP.

5.1 Preliminary Considerations

Several frameworks and programming languages were examined prior to beginning this task. The main requirement for the chosen framework is to be able to produce an application that runs on the target platforms. As the program will mainly be driving OpenVPN via shell commands, it should have a good support of those.

Selected languages are exposed here.

5.1.1 REALbasic [13]

REALbasic is both a modern, object-oriented implementation of BASIC with a VisualBasic-like syntax and an IDE that runs on Mac OS X and Windows; it also compiles binaries for Linux systems.

Designing a GUI with REALbasic is very convenient and easy. The fact that its output runs natively on the target platforms without the need of any other prior installation makes this option very interesting. REALbasic offers a *Shell* class which lets a program interact with a process started from the command line, synchronously or asynchronously. The existence of threads could have made it possible to run OpenVPN while still maintaining a responsive GUI.

However, the described extended shell support is only available for Mac OS X. Linux and Windows only get the synchronous *Shell.Execute* command, which only takes a single parameter: the command to be executed. There is no way to find out what the output of the started process is; the only available result data is the exit error code of the process, if any. Moreover, REALbasic is not free and a licence for the cross-platform IDE would have to be purchased.

REALbasic is thus not suited for the Connection Assistant.

5.1.2 C# and .Net

Microsoft's C# language and .Net framework provide basic shell interaction possibilities. It is for instance easy to start a new process, using the *System.Diagnostics.Process* class. However, as with REALbasic, it is impossible to read the output of the process and thus to determine if it is running correctly in the case of OpenVPN.

Platform compatibility is also an issue with .Net. The output of the standard compiler does produce bytecode and not machine instructions, so it is theoretically possible to implement the virtual machine specification on Mac OS X and Linux. Actually, the Mono project is going this way.[13] However, this would require extra installers and the solution cannot be considered fully reliable at the time of writing; moreover, as with REALbasic, a licence for the development environment, Microsoft Visual Studio .NET, would have to be purchased.

This is why C# was put aside, too.

[13] More information about Mono: [14].

5.1.3 Java

Java is the most important cross-platform language. Virtual machines exist for a variety of platforms, including of course our target platforms—and others, including Solaris, also supported by OpenVPN.

Java has a very good shell interaction support through its *java.lang.Process* class, and allows input and output streams to and from the running process. Java Threads make it easy to maintain the GUI while OpenVPN's output is still correctly handled.

Java's downside is that a virtual machine is needed on the clients' machines. With Mac OS X, this problem is not an issue, as Java comes preinstalled with it; it has to be installed on Linux and Windows though. Solutions trying to solve this problem can be explored: native executable builders, or tools to bundle a Java virtual machine with the compiled application.[14] See Section 5.4 for more information about Java executable bundling. Furthermore, distributing a Java virtual machine installer (JRE) does not seem to cause licensing problems.

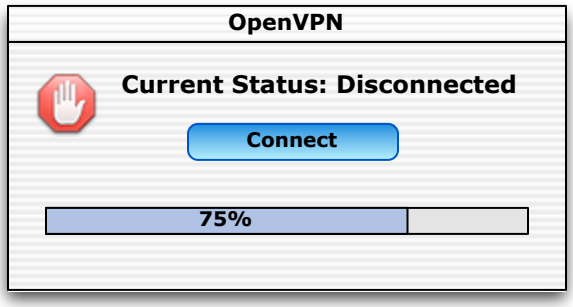
Java was thus selected as development platform.

5.2 Short Design

5.2.1 GUI Prototyping

The graphical user interface must be a single status window. It has to indicate the current connection status in an unambiguous way. A button must let the user initiate the connection, interrupt an ongoing connection attempt or disconnect from the VPN, depending on the current status.

During the connection, a progress bar must monitor the ongoing operations.



It is to be kept in mind that the GUI is the only useable way for standard users to deal with a running application. Therefore, it would be bad design if it was possible to leave the GUI with OpenVPN still running, because the user would not know how to properly disconnect.

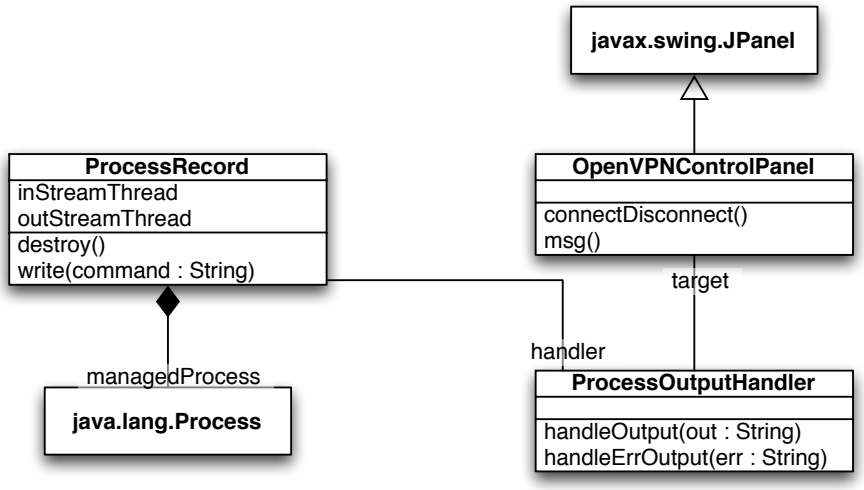
It would be a good idea that an option offers the possibility to observe the textual output of OpenVPN as an option, not only for more experienced users, but also for debugging purposes.

Platform-specific UI decorations, like a Windows Tray icon or a Mac OS X customized Dock menu, would be a plus. The former can be achieved using the TrayIcon API [17], a Java library which uses native code.

[14] Java Launcher, for instance, allows to create double-clickable .exe Windows applications. See [15]. JSmooth does a similar job; see [16].

5.2.2 Class Diagram

Here is a simple class diagram of the Connection Assistant. Only public methods and attributes have been included. All omitted multiplicities are 1..1.

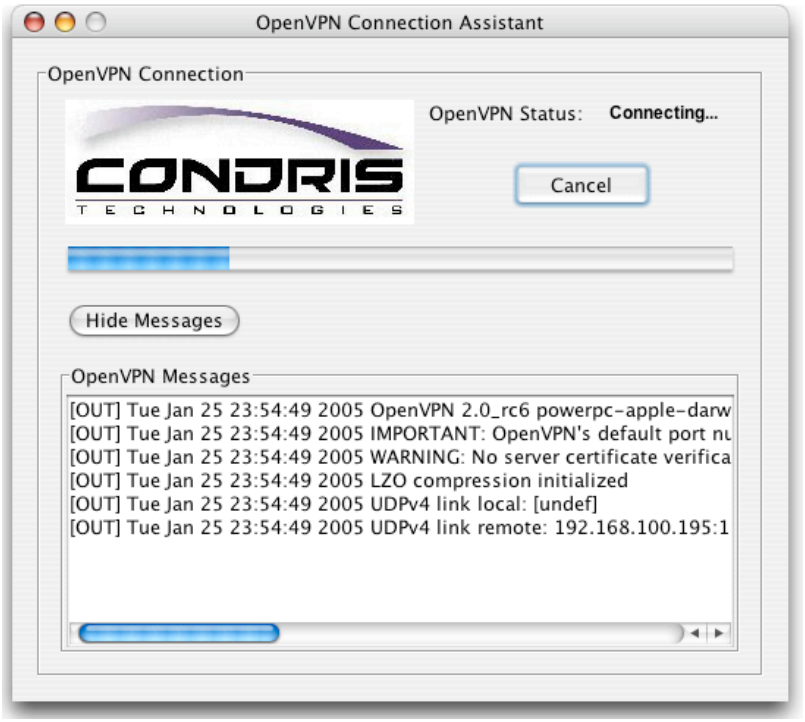


The *ProcessRecord* object contains two threads that look for available output from the *java.lang.Process* object—the OpenVPN instance. They then pass this output to their handler by calling *handleOutput()* or *handleErrOutput()* depending on the kind of output from the process. This handler must be passed as parameter to the *ProcessRecord*'s constructor: it is created in the *OpenVPNControlPanel* object.

This object is both the view of the application and part of the controller. The *connectDisconnect()* method is called when the user pushes the main button. It causes either a start of a new OpenVPN instance or the interruption of the current OpenVPN execution. *msg()* is called by the *ProcessOutputHandler*.

5.3 Implementation

In order to better visualize it, here are two screen shots of the final window on Mac OS X.



5.3.1 *ProcessRecord* and *ProcessOutputHandler*

The threads responsible for watching the process's output have been implemented mainly as a loop polling for new output lines every *THREAD_SLEEP_DURATION* milliseconds, set to 50 in order to still get a responsive output rate when OpenVPN is just starting and producing a lot of indication messages.

The threads also watch a synchronized private variable *goOn* after each output in order to know whether the monitoring must go on. When the *destroy()* method is called, this variable is set to false and the threads stop and die. In order to ensure proper termination, they are *join()*'ed in *ProcessRecord*'s overridden *finalize()* method.

5.3.2 *OpenVPNControlPanel*, Launching OpenVPN

The class *OpenVPNControlPanel* is responsible for the main part of the graphical interface. In particular, it has to start and stop OpenVPN, and watch for the process's output in order to update its progress bar as well as to inform the user about connection establishment or possible failures.

In order to allow changes in the configuration of the client, it was decided to run OpenVPN by passing it a configuration file rather than hard-coding the parameters in the command line. This configuration file should conveniently be stored in a subfolder of the application's folder, named for instance "config". The runtime path of a Java application can be determined using the *new java.io.File("").getAbsolutePath()* command, from which subdirectories and subfiles can easily be found.

Basically, OpenVPN will not only need the configuration file, but also the private key and the certificates, which are also wanted to reside in the mentioned subfolder. For that purpose, the command-line argument `--cd` is used, with the "config" path as parameter.

An important restriction in order for the *openvpn* command to be successful, is that the OpenVPN directory is in the *PATH* environment variable. An alternative is to use its complete path, for instance */usr/local/sbin/openvpn*, but this path is not known in advance and should not be hard-coded.

5.3.3 Monitoring The Connection Progress

In order to draw an adequate progress bar, it would be ideal to know how much time each step of the connection establishment lasts, and to adequately divide the progress bar into n subparts, whose length should be proportional to its average execution time. This is unfortunately not possible because the Connection Assistant cannot directly have access to information about what exactly OpenVPN is currently doing. The next best solution is to try to identify important connection steps looking at OpenVPN's output and to check for them before outputting them via the *msg()* method of the *OpenVPNControlPanel* class.

The four main identifiable steps during the connection establishment roughly correspond to the output of lines containing the following character strings:

```
OpenVPN
Peer Connection Initiated
TAP-WIN32 device [OpenVPN] opened
Initialization Sequence Completed
```

If the string number i is found in a line output by OpenVPN, the progress bar is advanced to position i and the thread now looks for string $i + 1$ to be output.

In order to be able to adapt to newer versions of OpenVPN and to its possibly changed messages, all these character strings are stored in an external file and are not hard-coded.

5.3.4 Connection Errors

A list of errors can happen during the connection attempt. In most cases, OpenVPN can be stopped and the user shown a warning message describing the error. Detecting such an error should thus have the same effect as clicking the "Cancel" button during the connection establishment.

As OpenVPN-style error messages can be rather hard to understand, they are stored together with a more descriptive version of the error in a hash table. Whenever an error string is found in OpenVPN's output, the corresponding descriptive version is retrieved and shown to the user as standard error message in a modal window. OpenVPN is subsequently stopped.

During test phases, a few typical error messages have been identified. They are listed below with their interpretations. Not all possible errors could be spotted; fortunately, OpenVPN always outputs the "Exiting." message if something goes wrong. This provides an easy way to tell that the connection attempt has definitely failed.

`command not found`
⇒ *openvpn* is not in the PATH.

`All TAP-Win32 adapters on this system are currently in use`
⇒ Another copy of OpenVPN is running.

`Cannot allocate TUN/TAP dev dynamically`
⇒ OpenVPN must be run in root mode in Mac OS X to do that.

`TLS handshake failed`
⇒ Happens after a one-minute timeout. Can mean that the server is not responding or that there is a problem with private keys or certificates.

`Connection reset by peer`
⇒ The communication partner is not running OpenVPN.

`Host is down`
⇒ The server is not responding.

`Exiting`
⇒ Some other error has occurred.

5.3.5 Internationalization

The graphical interface must display a variety of character strings: captions, labels, buttons, error messages. In order to make the internationalization of the Connection Assistant easier, these strings should not be hard-coded but rather stored in an external file and loaded at runtime, just like the connection messages or error messages from OpenVPN.

Java provides a simple mechanism for internationalization: Properties files. Properties files are text files containing key/value pairs, in the form *key=value*. Multiple properties files can be located in the same folder and loaded by a *ResourceBundle* object according to the default or current locale. The default properties file can for instance be called *ApplicationStrings.properties*. Language-specific localizations would then be named *ApplicationStrings_de.properties* for German or *ApplicationStrings_fr.properties* for French for instance.

In order to keep the code used to retrieve the strings short, a class *Res* was created, with a static method *get(key : String) : String* returning the value associated with the passed key. The class looks for the default resource bundle at startup and calls *System.exit()* if it can't be found, alerting the user (in English, using a hard-coded string) that the properties file is missing. However, as properties files will be embedded in the final Jar files, this problem should never happen.

5.3.6 Platform Caveats

Calling *ProcessRecord.destroy()* normally also destroys the managed *java.lang.Process* object by calling the *destroy()* method on it too. This works fine on Mac OS X and Linux. However, on Windows, the process continues to live, and OpenVPN is never terminated correctly. Several workaround have been tried. A temporary solution was found using the following command to kill OpenVPN:

```
taskkill /F /IM openvpn.exe
```

Of course, this solution is not very elegant, because it causes an abnormal termination. Moreover, it may kill another running instance of OpenVPN... which might not be a limitation in itself after all, because the TAP-Win32 driver seems to forbid two running instances, which would make the use of several running instances of OpenVPN impossible anyway.

Another problem on the Windows platform is that the previous default gateway does not seem to be reset automatically after VPN disconnection. It seems to behave sporadically badly. But it can be easily set using this shell command:

`route ADD 0.0.0.0 MASK 0.0.0.0` *`gateway-addr`*

On the other hand, a lot of routing information can be retrieved using the command

```
route PRINT
```

It is therefore possible to parse its output and remember the current default gateway before starting OpenVPN and to restore it after disconnection using the first command.

5.4 Deployment Notes

Although the full deployment process is beyond the scope of this project, a few ideas or solutions are mentioned here.

The problem of Java applications is that, on many platforms, they are not first-class citizens: they have to be started from the command line and/or the path to a currently installed Java Virtual Machine must be known, etc., which could lead to unpredictable results or frustration for the users.

Ideally, they would be presented to users as a traditional, platform-native application. Of course, this goes against the cross-platform need that was expressed in 3. In most cases, platform-specific tools or solutions are needed.

5.4.1 InstallAnywhere [18]

InstallAnywhere is a commercial product by Zero G. It cross-compiles professional-looking installer solutions for a variety of platforms and is often chosen by big Java applications developers because it provides a high level of abstraction over the platform-specific installation procedure, allowing them to keep as cross-platform in their design and realization as they have been during Java development, while still allowing to fine-tune specific details, like Windows registry entries or file associations.

Perhaps most interesting is InstallAnywhere's ability to bundle a Virtual Machine with a Java application, making the whole package appear and behave just like a native application. JVMs can be individually downloaded [19] and bundled with application with apparently no licensing issue. InstallAnywhere would have been the solution of choice if its price was not as high as $2999.

5.4.2 JSmooth [16]

JSmooth is targeted at Windows users. It takes as input a Java JAR application and produces a traditional EXE file out of it. Application and VM parameters can be set; it also allows to automatically search for already installed JVMs and allows to give different search priorities for specific user folders, environment variables, registry key values and application subfolders possibly containing a JVM—this allows to bundle a JVM in one of the application's subfolders, thus making it independent from the application's main folder. If no JVM is found, a prompt to download one can be displayed. Finally, a custom icon can be set for the executable.

The result is an almost native Windows application. As JSmooth is distributed under the GNU Public License, it was chosen for final Windows JAR bundling of the Connection Assistant.

5.4.3 Jar Bundler

Jar Bundler does for Mac OS X a similar job as JSmooth for Windows: it wraps a traditonal JAR application into a native application with a custom icon, preset classpath definition, application and VM parameters, and standard double-clickable form. As every copy of Mac OS X comes with a JVM, there is no need to bundle one with the application as the problem is non-existent. Jar Bundler comes with the free Apple Developer Tools.

Further Mac OS X-specific options can be set using Jar Bundler, like the possibility to move menus from Java frames to the standard Mac OS X menu bar, which is much more adequate and professional-looking for long-term users of the Mac platform. Mac OS types and creators

can also be set, thus allowing documents with a specific type or extension to be automatically linked to the created application without the need of any other file association definition.

The created application takes the form of a Mac OS X application package—that is, an icon behaving and reacting just like an normal application, but being actually a folder. This makes it easy to conceal additional support or localization files from the users. The working directory can be set to inside the application package, thus allowing to transparently benefit from this additional encapsulation from the Java code.

6 Configuration File Generation

In this section, the problem of automatic on-the-fly configuration file generation will be treated. This problem is not only present with OpenVPN's configuration files, but also in the context of a bigger solution, whose general architecture elements are shortly mentioned here.

6.1 Architectural Context

A headless machine on Linux is acting as a gateway in a network structure (this is also the machine acting as the OpenVPN server). A set of Java Server Pages[15] pages coupled with a Hibernate [21] database conveniently allow to configure its behaviour. Each JSP corresponds to a *Section* object in the database. Each section has several parameters—mostly defined as character strings.

The machine has two Ethernet adapters corresponding to the Local Area Network which it serves and to the Wide Area Network from which internet connectivity is gained. The various sections define options such as WAN connection mode, WAN Domain Name Servers, NTP server address, LAN DHCP server mode, DHCP address range, etc. Upon a database commit of one of the sections, a command-line tool mapping the parameters to Linux configuration files or options in configuration files and/or Linux services to be started or stopped will be launched with the modified section name as parameter.

This part deals with the design and implementation of this tool.

6.2 Short Design

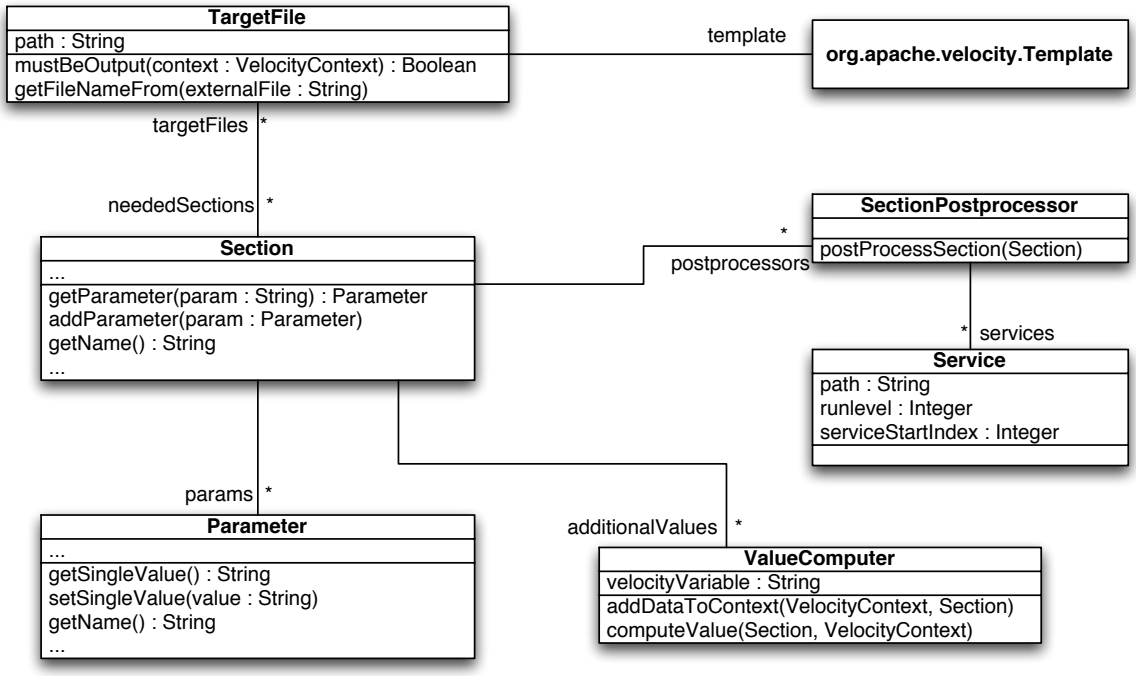
After preliminary considerations, the following has been pointed out:

- A section may cause changes in several files;
- A file may need parameters from different sections in order to be complete;
- A section may cause one or more services to be started, restarted or stopped;
- Certain file paths or names depend from the environment and may not be known in advance;
- Some parameters needing to be written to the files cannot directly be retrieved as section parameters but must be computed from several other parameters, possibly coming from different sections;
- Depending on some parameters, certain files need not be written at all.

The Velocity library will be used. Velocity is a Java engine producing a string output based on template files containing variables. The *VelocityContext* object (similar to a map) acts as link between the template variables and their actual values retrieved by Java code.

Here is a class diagram constructed with the Velocity facilities in mind proposed to model the problem under these considerations. All omitted multiplicities are 1..1.

[15] For more information and a tutorial on JSPs, see [20].



A few objects are further detailed:

- A *TargetFile* represents a file to be written. It has its own *Template* and needs one or more *Sections* in order to be written. It can read its file name when it must be dynamically determined using the *getFileNameFrom()* method. It knows whether it has to be written or not depending on the passed *VelocityContext*.
- A *Section* has its set of *Parameter*s and should know which *TargetFile*s it pertains to. It is also linked to a list of *ValueComputer*s, used to produce additional values needed by one of its *TargetFile*s but not present in its set of *Parameter*s.
- *ValueComputer*s are linked to a particular *Section*. They know the name of the Velocity variable they are linked to, and know how to compute its value based on other values of their *Section* or other values already set in the *VelovityContext* using their *computeValue()* method.
- *SectionPostprocessor*s are usually used to start or stop services. Their *postProcessSection()* method is invoked after all files of the *Section* they are linked to have been written; it can decide what action to perform based on parameter of the *Section* which is passed to it.
- Finally, a *Service* object models a Linux service, with some required parameters whose usage is explained later.

In real life, a function *loadSectionFromDB(sectionName : String)* is provided and used to retrieve individual *Section* objects from the Hibernate database.

6.3 Implementation

6.3.1 Java Implementation

The *Section* and *Parameter* objects are already defined, it is thus impossible to implement exactly the above class diagram because the additional relations between a *Section* and the

other auxiliary objects that have been defined here would be missing.

In practice, maps (*java.util.HashMap*) were used to simulate the *1:n* links; two maps were used to simulate the *n:m* link between *Section* and *TargetFile*. The names of the sections (resp. the *TargetFile*s) were used as keys pointing to lists (*java.util.LinkedList*s) of objects at the other end of the association.

In order to better group these additional links, they have all been defined in a new class, *Dependencies*, containing static initializers to create the dependencies and the following public methods:

- *getFilesForSection(section : String) : List<TargetFile>*. This method consults the map responsible for links from a section to a file list and returns them. It is used by the main procedure to know which files must be written for the passed section.
- *getSectionsForFile(file : TargetFile) : List<String>*. This is the counterpart to the previous method.
- *getNeededSectionsForSection(section : String) : Set<String>*. This returns all sections needed to write all files linked to the passed section. This method is called in order to know all sections that will have to be retrieved from the database in order to be put into the *VelocityContext* prior to generating the files from the templates. The sections are returned in a set rather than a list in order to avoid getting twice the same section, considering same auxiliary dependencies for two different *TargetFile*s.
- *postProcessContextForSection(context : VelocityContext, section : section)*. This method loops through all additional *ValueComputer*s for a given section and adds their value to the passed context.
- *postProcessSection(section : String)*. Loops through all *SectionPostprocessor*s for a given section. This is called after all files have been generated and allows services to be started or stopped, or additional commands to be issued.

The class also contains a few other methods in order to execute a shell command and get its result (see discussion on how to get OpenVPN's output)[16], convert an IP address as string to an IP address as integer and back, and to start and stop services.

In order to make service activation persistent across reboots, a symbolic link has to be created; similarly, symbolic links pointing to services having to be stopped must be deleted. The generic path to a service that will be started during Linux startup is

`/etc/init.d/rcx.d/Syyservicename`

where *x* is the Linux run level of the service, *yy* the two-digit service index, and *servicename* the actual name of the service, like *dhcpd* or *xntpd*. That is why the run level and service index is also stored in the *Service* object.

6.3.2 Velocity Templates

Velocity templates take the form of simple text files with the addition of variables. Velocity variables have the form *$variablename*. When producing the output files, their content is looked up in the passed *VelocityContext*. These variables can represent any Java class and can be manipulated with all their Java instance methods (which Velocity looks up and invokes using reflection). Furthermore, Velocity makes accessing the bean properties of a class easier by translating, for instance, *$variable.Property* into *$variable.getProperty()*.

[16] See [22] for a discussion about Java's *Runtime.exec()* method, its different parameter types and its caveats.

Variables can be manipulated using Velocity directives. Directives are represented by text lines starting with #. Comments are started with ## and last until the end of the current line. Among other directives, worth mentioning are:

- *#set($variable = expression) ## variable assignment*
- *#if(expression) ## conditional text insertion or directives*
  then-part
  #else
  else-part
  #end
- conjunction and disjunction in boolean expressions are expressed, as in Java, with && and ||, respectively.

6.4 Deployment

The deployment of this command-line tool is pretty straighforward. However, a few points are worth mentioning.

- The tool has to erase system files and replace them with newly-generated ones; it must therefore be run with root privileges.
- Velocity will be unable to find templates located in a JAR file if started with default parameters. In order to be able to package everything including templates in a single file, an alternative resource loader must be specified to Velocity. This is done by passing a *java.util.Properties* object while initializing Velocity with the following key/value pairs:
  - `resource.loader = classpath`
  - `classpath.resource.loader.class = org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader`

7 Conclusion

The number of wireless access points is constantly growing all over the world; the wi-fi market is seeing a considerable expansion. Among millions of access points spread worldwide, about 70% of them are unprotected and about 29% use WEP as security protocol. The remaining 1% works with WPA. Because we have seen that WEP cannot be considered secure, there is clearly still a lot to be done in this field regarding security and access control (the latter of which WEP does not provide at all).

During this project, we chose to work on a VPN solution rather than trying to implement WPA. As far as Mobile Device adapters are concerned, OS-level support is needed. Windows XP and Mac OS X, for instance, provide this support. WPA support in Access Points must also be provided by manufacturers, which is not always the case. Finally, open source implementation of software running on WPA's Authentication and Authorization server exist, but do not always work with WPA Access Points or Mobile Devices. WPA is generally not supported well enough in order to be suited for this project. Let us recall that this project will have an immediate practical use in the context of a bigger industrial project: this imposes the condition that the realised solution has to work and have a maximum compatibility.

Moreover, it must be noted that the scope of our solution is not identical to that of a WPA solution. The full WPA specification includes, on one hand, various protocols between Mobile Devices and Access Points, and on the other hand between Access Points and AAA server (i.e., Authentication, Authorization and Accounting server). Our VPN solution focuses on encrypting communication between Mobile Devices and Access Points. The VPN server software runs on an intermediate communication entity between the Access Points and the AAA server. The VPN solution shows its limitations compared to WPA when it comes to more flexible access control capabilities. Another reason for choosing VPN was to provide a temporary solution that could be used until we can achieve a full implementation of WPA.

Our choice of cross-platform, royalty-free VPN software led us to the use of OpenVPN, an open source command-line VPN solution. In order to increase the usability of OpenVPN for end users, we developed a simple graphical front end to the command-line tool and detailed its implementation in Java. Finally, we explained our use of Velocity framework in the context of automatic configuration file generation in relation with a object-oritented database containing the values needed to generate the files.

Acronyms

API Application Programming Interface

CA Certification Authority

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DHCP Dynamic Host Configuration Protocol

GUI Graphical User Interface

GNU GNU's Not Unix

GPL GNU General Public License

HTTP Hypertext Transfer Protocol

HTTPS Secure version of HTTP

IDE Integrated Development Environment

IEEE Institute of Electrical & Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IPSec Secure version of IP

JAR Java Archive

JRE Java Runtime Environment

JSP Java Server Page

JVM Java Virtual Machine

LAN Local Area Network

NTP Network Time Protocol9

PPT Point-to-Point Tunneling Protocol

RC4 Ron's Code 4. Symmetric cipher By Ron Rivest

RFC Request For Comment

SSL Secure Socket Layer

SMTP Simple Mail Transfer Protocol

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

VPN Virtual Private Network

WAN Wide Area Network

WEP Wired Equivalent Privacy

WPA Wi-Fi Protected Access

References

[1] Fluhrer et al. on WEP Security
http://www.golem.de/0108/15270.html

[2] WEP Security
http://www.experts-exchange.com/Security/Q_21090869.html

[3] FreeRADIUS
http://www.freeradius.org/

[4] Comments on the PPTP Protocol
http://www.schneier.com/pptp.html

[5] Comment on vTun security problems
http://www.off.net/~jme/vtun_secu.html

[6] Description of the Transport Security Layer Protocol, RFC2246
http://www.ietf.org/rfc/rfc2246.txt

[7] Description of the security layer collision problem
http://sites.inka.de/sites/bigred/devel/tcp-tcp.html

[8] TLS Sequence Diagram
http://www.eventhelix.com/RealtimeMantra/Networking/SSL.pdf

[9] TLS Key Derivation
http://www.faqs.org/rfcs/rfc2716.html

[10] OpenSSL
http://www.openssl.org/

[11] OpenSSL Certification Authority functions
http://www.openssl.org/docs/apps/ca.html

[12] More about Diffie-Hellman
http://www.rsasecurity.com/rsalabs/node.asp?id=2248

[13] REALSoftware's REALbasic
http://www.realbasic.com/

[14] The Mono project
http://www.mono-project.com/

[15] Java Launcher
http://www.syncedit.com/download_javalauncher.html

[16] JSmooth
http://jsmooth.sourceforge.net/

[17] Java TrayIcon API
http://jdic.dev.java.net/documentation/incubator/tray/

[18] InstallAnywhere
http://www.zerog.com/products_ia.shtml

[19] InstallAnywhere Virtual Machine packs
http://www.zerog.com/goto/vmpacks/

[20] Tutorial on Java Server Pages
http://www.jsptut.com/index.html

[21] Hibernate API
http://www.hinernate.org/

[22] Discussion on Java's *Runtime.exec()* command
http://www.mountainstorm.com/publications/javazine.html