Published on **O'Reilly** (http://www.oreilly.com/)
http://www.onlamp.com/pub/a/security/2004/10/21/vpns_and_pki.html
See this if you're having trouble printing code examples

Deploying a VPN with PKI

by Scott Brumbaugh
10/21/2004

This article is the second of two focusing on tunneling VPNs and Public Key Infrastructure (PKI). The first article provided a simple overview of the technology for people without a deep computer security background. This article is a walk-through tutorial that implements the concepts covered previously. We assume that the reader has some knowledge of VPN and digital certificate basics and wants to see exactly what it takes in practice to extend a private network by deploying a secure VPN. This tutorial presents code examples that will require novice- to intermediate-level system administration skills to reproduce, but anyone should be able to follow along with the discussion.

The tutorial implements a certificate-based security infrastructure using OpenSSL and uses this to secure both OpenVPN client and server endpoints. We will highlight two great new features to appear in OpenVPN-2.0 (now in beta) that will make it a good choice for any VPN--single-instance server mode and certificate revocation list support.

Deploying a VPN and Public Key Infrastructure

In part one of this series, we covered VPN and PKI
at a conceptual level. We discussed the benefits that VPN technology offers to a business network and touched on three different techniques or protocols that can actually implement a VPN tunnel. These were PPTP, L2TP/IPSec, and the relative newcomer, OpenVPN. An advantage of OpenVPN is that it encapsulates network traffic inside of UDP or TCP packets, which improves the odds of it working when deploying a VPN across unknown third-party network equipment and the Internet as a whole.

The previous article also talked about PKI and loosely defined the term as a system for issuing, publishing, and revoking digital certificates. These digital certificates are simple data structures containing, at the minimum, a subject's name, his public key, and the name of an issuer who has verified the subject's identity. When the issuer digitally signs the certificate and attaches the signature, the data structure becomes a digital certificate. The public key inside of the certificate can encrypt any message, producing a message decipherable only by the named subject in the certificate using the closely guarded private key.

For our step-by-step description of establishing connections across a VPN tunnel, secured with PKI, we will focus on two separate software packages, OpenVPN-2.0 (beta 8) and OpenSSL 0.9.7d. This version of OpenSSL was the latest stable version of the package at the time of writing. The OpenVPN beta version includes a couple of new features that really make it quite acceptable for a business VPN, namely, Certificate Revocation List (CRL) support and single-instance server mode. We'll use a generic GNU/Linux system to host both the PKI and VPN server endpoints. Our sample client configuration will be platform-neutral, meaning that the same client configuration should work on any supported OS, including Linux and the BSDs, Solaris, Mac, and Windows. Setup and operation for both OpenSSL and OpenVPN is similar on any supported platform.

First, we will configure OpenSSL to act as our root Certificate Authority (CA) by creating a self-signed certificate that will sit at the top of our trust hierarchy. Then, we will create certificate requests for our clients and actually issue certificates. Next, we will use OpenSSL's built-in test framework to make sure everything works smoothly before we complicate things by throwing a VPN into the mix. After running the OpenSSL test framework, we will configure both the OpenVPN server and client endpoints. Finally, we will demonstrate a CRL by revoking a user certificate, generating a CRL containing the revoked certificate, and effectively terminating the user's access to our VPN.

OpenSSL

OpenSSL is primarily a library of cryptographic functions that provides an extensive crypto API to programmers. However, it also includes a shell tool that exposes that API to users and batch scripts. Start the shell by typing `openssl` at the command line. From there, you can type commands at the `OpenSSL>` prompt.

```
[admin@tamarack admin]$ openssl
OpenSSL> version
OpenSSL 0.9.7d 17 Mar 2004
OpenSSL>
```

You can also issue OpenSSL commands in batch mode. In batch mode, each OpenSSL command executes separately. Shell scripts usually use this approach; we'll do the same in this tutorial:

```
[admin@tamarack admin]$ openssl version
OpenSSL 0.9.7d 17 Mar 2004
[admin@tamarack admin]$
```

Of the many commands provided with the OpenSSL shell, this tutorial will only cover five:

- `ca`: Certificate Authority management
- `req`: Certificate request management
- `verify`: Certificate verification
- `s_server`: Secure server test mode
- `s_client`: Secure client test mode

The OpenSSL Configuration File

When installing an OpenSSL Certificate Authority, we need to provide a master configuration file. This file contains default parameters for all OpenSSL commands. Compiling the library hard-codes its default location; its name is *openssl.cnf*. To find the default location of this file, use the `version` command with the `-d` option:

```
[admin@tamarack admin]$ openssl version -d
OPENSSLDIR: "/home/admin/install"
[admin@tamarack admin]$
```

Once you know where to look, edit this file with a text editor. The OpenSSL distribution includes a heavily commented example configuration file, but it's more complex than the simplified version we will work with in this tutorial. Hopefully, by paring the configuration down to a minimum, we will simplify our deployment without sacrificing security. Specifically, our configuration file removes many of the certificate extension definitions that appear in the prepackaged OpenSSL configuration.

It's probably a good time to mention these extensions. Part one of this series listed the *minimum* amount of information that must appear in a digital certificate: a subject name, the subject's public key, and the name of an issuer. Certificates that limit themselves to this minimum amount of information and conform to the X.509 encoding format are X.509v1 (version 1) certificates. Presently, the current standard specifies additional information that can appear in the certificate to increase security and aid applications that work with them. Most certificates used with Internet-enabled applications currently conform to the X.509v3 (version 3) specification and include some extensions. OpenSSL can produce either v1 or v3 certificates depending on the configuration file settings.

In this tutorial, we will configure OpenSSL to produce v3 certificates and include the `basicConstraints` extension to limit how people can use the certificates
that we issue, as this is the recommended configuration. Here is the configuration file that we will use:

```
# openssl.cnf
#
# OpenSSL example configuration file.
#
# This configuration file has been derived from the original
# example file included with the OpenSSL distribution.  It
# has been edited mostly to eliminate extensions in order to
# simplify it for the purpose of an online tutorial.  It has
# also been reformatted to limit the maximum line length to
# 60 characters for online publication.

################################################################
# This section will configure the ca (Certificate Authority)
# command.  We will use the ca command to sign user
# certificates and periodically generate CRLs.
################################################################

[ ca ]
default_ca      = CA_default      # The default ca section

[ CA_default ]

dir             = /home/admin/CA-DB  # Top
```

```
crl_dir         = $dir/crl        # The crl location
database        = $dir/index.txt  # Database index file
new_certs_dir   = $dir/newcerts   # Location for new certs
certificate     = $dir/cacert.pem # The CA certificate
serial          = $dir/serial     # The next serial number
crl             = $dir/crl.pem    # The current CRL

# All DNs need to be unique.  This is the default behavior
# but due to an OpenSSL library bug in the 0.9.7d release,
# if we don't supply this redundant definition here we will
# see a cryptic message when signing certificates.
unique_subject  = yes

# The CA private key
private_key     = $dir/private/cakey.pem

# Private random number file
RANDFILE        = $dir/private/.rand

# Issued certificates will be valid for 1 year
default_days    = 365
default_crl_days= 30

# Hashing function
default_md      = md5

# This assignment causes the extensions defined in the
# 'user_extensions' section to be included in any
# certificates that are signed using the ca command.

x509_extensions = user_extensions

# This sections describes the policy that will be enforced
# on a request to be signed, the subject organization name
# must match that in the CA certificate.  The request may
# contain an optional organizational unit name. The common
# name is assigned the policy format 'supplied' which means
# it must be present in the certificate request.

policy          = policy_any

[ policy_any ]

organizationName        = match
organizationalUnitName  = optional
commonName              = supplied

###############################################################
# This section configures the req (certificate request)
# command.
###############################################################

[ req ]

# The default key length and the filename that will contain
# a private key.  The public key will be contained in the
# certificate request.

default_bits            = 1024
default_keyfile         = privkey.pem
distinguished_name      = req_distinguished_name

# The makeup of our subject name
[ req_distinguished_name ]

organizationName                = Organization Name (eg, company)
organizationName_default        = Inyo Technical Services

organizationalUnitName          = Organizational Unit (eg, west)

commonName                      = Common Name (eg, YOUR name)
commonName_max                  = 64

# This assignment mimics that in the ca section but causes
```

```
# the following extensions to be included in a certificate
# request.  In our case these extensions will only be
# included when we self-sign a certificate using 'req
# -new -x509'.

x509_extensions = CA_extensions

#############################################################
# Extensions that will be added to certificates that are
# issued.  The 'user_extensions' sections contains
# definitions that will be included in certificates that are
# signed by this CA.  The CA_extensions section contains
# extensions that will be included when we create a
# self-signed certificate using the req command.
#############################################################

[ user_extensions ]

# CA:FALSE will not permit this certificate to sign other
# certificates.
basicConstraints        = CA:FALSE

[ CA_extensions ]

# CA:TRUE will allow this certificate to sign others.
basicConstraints        = CA:TRUE
```

There are quite a few things to note about this file. First, note that comment blocks divide it into logical sections. In this sample, those sections are `[ ca ]`, the configuration for the `ca` (Certificate Authority) command; `[ req ]`, the section containing items to include in a certificate request with the `req` command; and finally, a section defining some extensions. We only include one extension and assign it two different values depending on the certificate that we are creating.

Our self-signed CA certificate will include the extension `basicConstraints = CA:TRUE`. This allows a Certificate Authority to use the certificate to sign other certificates. When we issue certificates to users, we will use the extension `basicConstraints = CA:FALSE`, denying users the ability to create subordinate certificates and thereby extend the chain of trust beyond our control.

The start of the `[ ca ]`
section defines the directory hierarchy. This lets the OpenSSL library know where to find important files, and the definitions here should be clear. The assignment `x509_extensions = user_extensions` adds the extensions defined in the `[ user_extensions ]`
section to any certificates signed by this CA. As explained before, the only extension we will include is `basicConstraints`, which was explained previously. Next come the definitions of the certificate request subject's name-matching policy and Distinguished Name structure.

The `[ policy ]`
section defines the Distinguished Name components to allow in certificate requests. To understand this, we need to describe the encoding of subject names in X.509 certificates.

X.509 certificates encode subject names in a structure called a Distinguished Name (DN)--a sequence of name components called *Relative Distinguished Names*
(RDNs). This naming structure comes from a complex specification for methods and syntaxes defining communication between computer programs as seen in RFC 1274, which gives the attribute definitions for the DN components that we will use in our certificates. The `[ policy ]`
section specifies which RDNs or DN components must appear in a certificate request and which of these components must match the subject DN in the CA certificate prior to signing. Here, we have configured our CA to encode a subject's DN as follows:

- The subject name *must* contain an `organizationName` part that matches our CA certificate exactly--`match`.
- The subject name *may* contain an optional and arbitrary `organizationalUnitName`--`optional`.
- The subject name *must* contain an arbitrary `commonName` component--`supplied`.

In our application, we will try to be consistent and issue user certificates with a `commonName` component that matches the user's login ID, but OpenSSL will not enforce this.

The next section, labeled `[ req ]`, contains default configuration values for the `req` command. We will use this command to create certificate requests. The `default_bits` and `default_keyfile` definitions set the length of the private key and its location in the file system. More interesting here is the `[ req_distinguished_name ]` section, which defines default values for the RDN components that will make up our subject's Distinguished Name. Here we define the prompts that will appear at the terminal when we execute the `req` command. We assign a default value to our `organizationName`: `Inyo Technical Services`. Finally, the assignment `x509_extensions = CA_extensions` includes the extension defined in the `[ CA_extensions ]` section in a certificate request.

This extension assignment is similar to that in the `[ ca ]` section. The difference here is that these extensions will appear only in certificate requests, while extensions in the `[ user_extensions ]` section will appear in a certificate signed by our CA. By default, if an extension is present in a certificate request but does not appear in the `[ user_extensions ]` section, then the extension *will not* be present in the issued certificate. The extension `basicConstraints = CA:TRUE` will only appear in the certificate when we use the `req` command with the `-x509` option to create a self-signed root certificate.

The Certificate Authority Root Certificate

With the master configuration file taken care of we can now set up the directory structure to hold our CA. We choose a suitable location and make our directories and initialize the *serial* file.

```
[admin@tamarack admin]$ mkdir CA-DB
[admin@tamarack admin]$ cd CA-DB
[admin@tamarack CA-DB]$ mkdir crl
[admin@tamarack CA-DB]$ mkdir newcerts
[admin@tamarack CA-DB]$ mkdir private
[admin@tamarack CA-DB]$ echo "01" > serial
[admin@tamarack CA-DB]$ touch index.txt
[admin@tamarack CA-DB]$
```

Notice that we seed the file *serial*
with the first serial number. As we issue certificates, the number stored in this file will automatically increment so each certificate will receive a unique serial number. The file *index.txt* will record all certificates issued. Records in this file will map the subject DN in a certificate to its assigned serial number. Using *index.txt*, we can look up the serial number for any certificate using the subject's DN. This will become important when we decide that we need to revoke a certificate and need to find the original.

As defined in the configuration file, the *new_certs_dir*
will hold copies of all certificates issued by this CA. The filenames for these certificates will consist of the serial number assigned to the certificate by the CA followed by an extension.

With the directory structure in place, we can create our self-signed root certificate. This certificate will sit at the top of our trust hierarchy. We will use it to sign all of the rest. For our tutorial, we will only use a single layer of trust, but a more robust design would probably use more.

```
[admin@tamarack CA-DB]$ openssl req -new -x509 -keyout \
        > private/cakey.pem -out cacert.pem
Generating a 1024 bit RSA private key
.....................................................
writing new private key to 'private/cakey.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will
be incorporated into your certificate request.
What you are about to enter is what is called a
Distinguished Name or a DN. There are quite a few fields
but you can leave some blank For some fields there will
```

```
be a default value, If you enter '.', the field will be
left blank.
-----
Organization Name (eg, company) [Inyo Technical Services]:
Organizational Unit (eg, west) []:
Common Name (eg, YOUR name) []:Root CA
[admin@tamarack CA-DB]$
```

Note: here and in the code examples that follow, the output has been reformatted to a maximum width of 60 characters.

To clarify some potential confusion, the `-x509` option will cause the `req` command to output a self-signed certificate rather than a certificate request. We will only use this option with the `req` command one time when we create our root certificate.

User Certificates

Next, we create a certificate request for the OpenVPN server endpoint installed on our local network. Note that we use the `-nodes`
argument. This prevents the private key generated with the request from being encrypted and password-protected. We will need to guard the key carefully because of this. We don't password protect this one private key because later we may want to start our OpenVPN server automatically at boot time when there is no interactive user available to supply the password. We could enter the password into the boot script itself, but that would sort of defeat the purpose of having a password in the first place. Physical security (such as locking the door to the server room) is the best way to protect this non-encrypted private key.

```
[admin@tamarack admin]$ openssl req -new -nodes -keyout \
        > vpnkey.pem -out vpncert-req.pem
Generating a 1024 bit RSA private key
.....................................................
writing new private key to 'vpnkey.pem'
-----
You are about to be asked to enter information that will
be incorporated into your certificate request.
What you are about to enter is what is called a
Distinguished Name or a DN. There are quite a few fields
but you can leave some blank For some fields there will
be a default value, If you enter '.', the field will be
left blank.
-----
Organization Name (eg, company) [Inyo Technical Services]:
Organizational Unit (eg, west) []:
Common Name (eg, YOUR name) []:vpn.inyotech.com
[admin@tamarack admin]$
```

Now we digitally sign the server certificate using the root certificate we created initially. OpenSSL will retrieve the location of the root certificate from the configuration file, which is why it does not appear as a argument on the command line. This command will update the files *serial* and *index.txt* when it completes.

```
[admin@tamarack admin]$ openssl ca -out vpncert.pem \
> -in vpncert-req.pem
Using configuration from /home/admin/install/openssl.cnf
Enter pass phrase for /home/admin/CA-DB/private/cakey.pem:
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
organizationName      :PRINTABLE:'Inyo Technical Services'
commonName            :PRINTABLE:'vpn.inyotech.com'
Certificate is to be certified until Aug 18 22:50:07 2005
GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
[admin@tamarack admin]$
```

After creating the server certificate, we can create certificates for all of the clients to whom we want to give VPN access. The procedure is the same, except we don't specify the `-nodes` option with the `req` command. This will encrypt the private key, and OpenSSL will prompt for a password use when seeding the cipher.

OpenSSL Test Framework

Now, after we have issued a couple of user certificates, we can make sure that our procedures are all correct by taking advantage of the two test commands provided by the OpenSSL package. The programs `s_server` (secure server) and `s_client` (secure client) can exercise almost the entire library and their operation is straightforward.

Start an OpenSSL secure server session in one terminal window. Start an OpenSSL secure client session in another. The client will contact the server using the SSL/TLS protocol at localhost using port 4433. You will be able to type messages into the console hosting the secure client and see them appear at the secure server. It will be immediately obvious if your certificates are not correct or there is a problem with your OpenSSL library installation.

Here we start an OpenSSL secure server at the command line. For arguments, we include the server certificate and server private key. The argument `-verify 1`
causes the server to ask any connecting client to send a certificate for authentication. (Note that the output from these commands is more verbose than these trimmed code examples indicate.)

```
[admin@tamarack admin]$ openssl s_server -cert vpncert.pem \
> -key vpnkey.pem -verify 1
verify depth is 1
Using default temp DH parameters
ACCEPT
...
[admin@tamarack admin]$
```

Now, in another console window, we start an OpenSSL secure client using the command argument `-cert` to provide a certificate to send to the server for authentication. The `-key` argument gives the private key to use when encrypting messages and the `-CAfile` argument points to the root certificate.

```
[admin@tamarack admin]$ openssl s_client -CAfile \
> CA-DB/cacert.pem -cert client1cert.pem -key client1key.pem
Enter PEM pass phrase:
...
[admin@tamarack admin]$
```

When the connection attempt succeeds, you can send sample messages between the client and server by typing text into either secure endpoint. To quit the session, type `Q` in the terminal window.

Now we know that our certificates can encrypt messages passed between two OpenSSL applications. However, we have not yet made sure that we can use our certificates with any arbitrary X.509-certificate-secured application. Adding the `-WWW` option to the `s_server`
command will effectively create a secure web server that can serve any local file to a web-browsing client connecting using SSL/TLS. We will exercise this feature next.

In the current working directory, create a small HTML file containing text similar to the following:

```
<html>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

Give the file a name such as *hello.html* and then start a secure web server from the same directory as the file using the `-WWW` option.

```
[admin@tamarack admin]$ openssl s_server -cert vpncert.pem \
> -key vpnkey.pem -WWW
Using default temp DH parameters
ACCEPT
```

Start any modern web browser, such as Mozilla, Netscape, or Opera. Internet Explorer should also work, if you are following along with this tutorial on Windows. Enter the following URL in the address bar, noting that the protocol is `https` and not `http`:

```
https://localhost:4433/hello.html
```

The web browser should alert you to the fact that it does not recognize the subject in the certificate offered by the server. Most browsers will allow you to accept the authenticity of a server certificate temporarily for a single session. After clicking through any such warning messages, you should see the sample page appear in the browser window. A client can connect to the secure web server from a remote server, as well.

The `s_server` is an *extremely*
useful test tool. Secure applications using certificates can be difficult to debug, especially if you are not sure where a potential problem may exist. OpenSSL's test client and server let an administrator more easily isolate problems to either a network application or its supporting certificate infrastructure.

OpenVPN

After creating our certificate infrastructure and testing our server and client certificates, we are ready to set up the VPN itself. OpenVPN uses a secure protocol called Diffie-Hellman to negotiate authentication (see RFC 2631 for a technical overview). We need to generate a set of parameters to facilitate this in a file called *dh1024.pem*. This file only needs to exist at the server VPN endpoint.

```
[admin@tamarack admin]$ openssl dhparam -out dh1024.pem 1024
Generating DH parameters, 1024 bit long safe prime,
generator 2
This is going to take a long time
.....................................................+........
[admin@tamarack admin]$
```

Next, we need to configure our operating system for OpenVPN. This is the only platform-dependent configuration step in this tutorial. As we noted earlier, our target server platform is generic GNU/Linux. If you are installing OpenVPN on a Windows platform from pre-compiled binaries, the installer will automatically perform the following step. On our Linux system, we configure the `tun`
(tunneling) software driver. If it does not already exist, use the following commands to create the device interface file in */dev/net/* and load the `tun` kernel module.

```
[root@tamarack /]# mkdir /dev/net
[root@tamarack /]# mknod /dev/net/tun c 10 200
[root@tamarack /]# /sbin/modprobe tun
```

The `tun`
module allows the operating system kernel to redirect network packets between the tunneling network device driver and a user-space program, in our case OpenVPN. When an application wants to send a network packet out through the VPN, it will send it through the `tun`
device. The kernel will then pass the packet to OpenVPN, which will use functions from the OpenSSL library to encrypt the packet before sending it out a real network interface to the client. Applications will receive packets from the client in a similar way. OpenVPN will read the packet from the real network device, decrypt it using OpenSSL functions, and then pass the decrypted packet to the kernel that will redirect it to the `tun` device.

Server And Client Endpoint Configuration

A configuration file tells OpenVPN how to operate. There are many different variations possible. Here, we will set up separate server and client configurations. We want the VPN to operate in single-instance server mode. This allows all client connections to go through the same server port, and it makes configuration much easier. As mentioned before, this single-instance mode greatly reduces the burden on an administrator to support multiple clients connecting simultaneously. Previously, each client connection would require its own separately configured server instance.

In single-instance server (hereafter simply referred to as "server") mode, the `ifconfig` parameter works a little differently than the way it used to in older versions of OpenVPN. The first parameter is the IP address of the local end of the tunnel, but the second parameter is not the remote IP address. Instead, it is the address of the gateway interface that OpenVPN will use locally.

The `ipconfig-pool` parameter gives a range of IP addresses to distribute to connecting clients. The `route` parameter is the route that will be set up on the local network to direct packets out of the VPN, while the `push "route ..."` command is the network route to set up on the client. Here is the configuration file for our OpenVPN server endpoint.

```
# openvpn-server.conf
#
# Tunnel mode
dev tun

# Run as a single instance server
mode server

# Server endpoint appears first, followed
# by the gateway interface ip
ifconfig 10.1.0.1 10.1.0.2

# Range of IP addresses reserved for clients
ifconfig-pool 10.1.0.4 10.1.0.254

# route setup on the server
route 10.1.0.0 255.255.255.0

# route command pushed to the client
push "route 10.1.0.1 255.255.255.255"

# Specify tls-server for certificate exchange
tls-server

# Diffie-Hellman Parameters (tls-server only)
dh dh1024.pem

# Root certificate
ca CA-DB/cacert.pem

# Server certificate
cert vpncert.pem

# Server private key
key vpnkey.pem

# Check for revoked client certificates.
crl-verify CA-DB/crl/crl.pem
```

Notice that we use the `dh`
option to specify the file containing the Diffie-Hellman parameters that we created earlier. The `dh` option only needs to appear in the server's configuration file and not in the client's. Next, we list files containing the certificates that we will use, starting with the root. Finally, we indicate that we want OpenVPN to check our certificate revocation list before authorizing a client to connect to our private network by specifying the `crl-verify` option and assigning it the location of a current CRL. We will demonstrate the CRL verification feature later.

The client configuration file is a little simpler. The only configuration item that we have not seen before is `pull`, which complements the `push "route ..."` command we included in the server configuration. Here `pull` will set up the route that we want packets destined for our VPN to use by receiving it from the server.

```
# openvpn-client.conf
#
# Set tunnel mode
dev tun

# Hostname for the VPN server
remote vpn.inyotech.com

# This end takes the client role for
```

```
# certificate exchange
tls-client

# Certificate Authority file
ca cacert.pem

# Our certificate/public key
cert client1cert.pem

# Our private key
key client1key.pem

# Get the rest of our configuration
# from the server.
pull
```

The next step is to install OpenVPN on the client. This step will only be covered here by saying that installation of OpenVPN as a client is identical to installing it as a server. After installation, complete the process by distributing the following files to the client:

- The user's private key.
- The user's certificate.
- A copy of the root certificate (so the VPN client endpoint can verify the server).
- The client configuration file.

Before starting the server, we have one last step to perform. We need to initialize the CRL. The CRL will hold a list of user certificates that our CA has revoked. Even though we have not revoked any user certificates yet, our configuration still asks the OpenVPN server to check the CRL using the `crl-verify` option in the server configuration. If our VPN server cannot find a CRL, it will exit prematurely.

For now, we will initialize an empty CRL. Later, we will revoke a user certificate and update the CRL. To initialize the CRL simply ask the CA to generate one. (The cryptic message beginning `DEBUG[load_index]` comes from a harmless bug in this particular release of OpenSSL; ignore it.)

```
[admin@tamarack admin]$ openssl ca -gencrl -out \
> CA-DB/crl/crl.pem
Using configuration from /home/admin/install/openssl.cnf
Enter pass phrase for /home/admin/CA-DB/private/cakey.pem:
DEBUG[load_index]: unique_subject = "yes"
[admin@tamarack admin]$ cat CA-DB/crl/crl.pem
```

Now we are ready to use the following command line to start a VPN server endpoint on our local network. After executing this command, clients can connect.

```
[root@tamarack admin]# openvpn --config openvpn-server.conf
```

After the server starts, attempt to connect to it from a client. The command line at the client will be very similar, but remember to use the name of the proper configuration file. On Windows, right-clicking on the configuration file and selecting Start OpenVPN on this config file from the context menu is sufficient. Later, you will probably want to set up a shortcut on the desktop.

You should immediately be able to communicate over the VPN. If there is a problem, it is time to troubleshoot. In this case, you will be glad that you tested your certificate infrastructure using `s_server` and `s_client` so you can better isolate what may be wrong.

Of course, to use the VPN tunnel fully, you will probably want to make applicable additions to the client DNS and make sure that the routing table that OpenVPN automatically sets up is correct for your network. We have only provided a single host route from the client to the server. This tutorial will not cover configuring DNS and routing. Treat the `tun` device like any other network interface.

Verifying Client Access Against a CRL

The final task that we will tackle is reliably terminating VPN access for users when we no longer want them to connect to our network. To do this, we will revoke their certificates and update our CRL. To figure out what certificate belongs to which client, use the file *index.txt*, located in the certificate directory tree. Look in the file to find the record containing the `commonName` attribute assigned to the certificate to revoke (remember that we established a policy of assigning the user's unique login ID to the `commonName` attribute). This record will also contain the serial number that the CA assigned to this certificate.

Having identified the serial number, we can locate the client certificate under *newcerts* in our CA directory. This directory will contain a copy of each certificate issued by our CA. The file names contain the serial numbers assigned to these certificates.

In this example, we decide to revoke the certificate for *user3*. We issued this certificate earlier following the procedure described in the section *User Certificates*. After looking in *index.txt*, we find that the serial number assigned to the certificate with the `commonName` attribute matching *user3* is `04`. We then know that the correct certificate to revoke is *newcerts/04.pem*.

```
[admin@tamarack admin]$ openssl ca -revoke \
> CA-DB/newcerts/04.pem
Using configuration from /home/admin/install/openssl.cnf
Enter pass phrase for /home/admin/CA-DB/private/cakey.pem:
DEBUG[load_index]: unique_subject = "yes"
Revoking Certificate 04.
Data Base Updated
[admin@tamarack admin]$
```

The command has revoked the certificate and updated *index.txt* appropriately. Now we need to generate a new CRL that will contain this newly revoked certificate. Until we do this, OpenVPN will not know that we have revoked this certificate.

```
[admin@tamarack admin]$ openssl ca -gencrl -out CA-DB/crl/crl.pem
Using configuration from /home/admin/install/openssl.cnf
Enter pass phrase for /home/admin/CA-DB/private/cakey.pem:
DEBUG[load_index]: unique_subject = "yes"
[admin@tamarack admin]$
```

From this point on, *user3* will not have access to the VPN, as long as the OpenVPN server endpoint checks the CRL. Remember, whenever you revoke a user's certificate, make sure to regenerate the CRL.

OpenSSL also includes the `verify` command, which accepts the `-crl_check` option that will allow you to make sure that your certificate revocation list works. The `verify` command requires that the root certificate and CRL live in the same file. Before you exercise this command, create a new temporary CRL by concatenating it with your CA certificate.

```
[admin@tamarack admin]$ cat \
> CA-DB/cacert.pem CA-DB/crl/crl.pem > tempcrl.pem
```

Use the `verify` command with the `-crl_check` option to specify the CRL to use. Here's how to verify the revocation of the certificate for *user3*.

```
[admin@tamarack admin]$ openssl verify -CAfile tempcrl.pem \
> -crl_check user3cert.pem
user3cert.pem: /O=Inyo Technical Services/CN=user3
error 23 at 0 depth lookup:certificate revoked
[admin@tamarack admin]$
```

In order to try this with your own setup, revoke the certificate that you created for your client VPN endpoint and regenerate the CRL. You will find that you can no longer connect to the OpenVPN server from that client.

Conclusion

If you have followed along and experimented with the code samples provided, you should have an understanding of the basics of implementing a PKI using OpenSSL and a VPN using OpenVPN.

We have set up a certificate infrastructure including a root CA and issued user certificates. This included configuring the OpenSSL library to add a v3 extension into the certificates issued by the CA. We have also summarized the `s_server` and `s_client` test framework, which you can explore further through the manual pages included with the OpenSSL distribution. We demonstrated setting up both client and server VPN endpoints and provided certificates for both authentication and encryption. Finally, we have shown how to use a CRL to revoke user certificates and terminate access to our VPN.

This concludes our two-part series on VPN and Public Key Infrastructure. We hope you have found it useful. Of course, we need to extend special appreciation to the creators of the OpenVPN and OpenSSL software packages and the GNU/Linux operating system that provides the solid foundation that lets us all work and build great software together. If you have any questions, feel free to contact us.

Scott Brumbaugh has worked professionally as a software/systems engineer since 1987.

Return to the Security DevCenter