Introduction

In this lab, you will practice certificate management and configuration of an IPsec Virtual Private Network (VPN).

This document describes the tasks you will do in the lab and must not be viewed as a manual or tutorial (although rather detailed information is provided). You are expected to use the slides from the lecture, to read the documents available under Links in Its Learning and to make full use of the search engines on the web. See also Deliverables section at the end of this document.

The VirtualBox appliance from *Lab 1: Linux networking and firewalls* will be reused here. You should reboot any VMs you have started, so that iptables configuration is reset the original Default ACCEPT policy. The ACCEPT policy will simplify somewhat the debugging in the early stages of the lab.

If you have made persistent changes to the iptables rules (changes that cannot be undone by a reboot) you can restore the VM to the state saved in the *Complete* snapshot. However, if you restore the VM, make sure to backup your changes (in particular the firewall.sh script) since **all** changes to the VM will be gone. Alternatively, VirtualBox will ask you if you want to take a snapshot of the current state before restoring the *Complete* snapshot. This is also an acceptable solution to backup your changes.

- You will start by configuring Server A to act as a certification authority (CA). Your CA system will be based on OpenSSL. The other VMs can be turned off during this stage.
- After that you will configure a *transport mode* IPsec VPN with *pre-shared key (PSK)* authentication. The transport mode VPN will require both Server A and Server B.
- In the next step, you will modify the VPN to use *certificate-based authentication* with the certificates signed by your CA.
- At this point you will modify the certificate-based VPN to use *tunnel mode* to protect Site A and Site B. Towards the end you will need both Client A and Client B, in addition to Server A and Server B.
- In the final stage, you will reuse the firewall.sh script from Lab 1.1 to configure the firewalls on Server A and Server B to default DROP policy. The firewall must allow traffic to the Internet and simultaneously ensure that all traffic between Site A and Site be is carried over the VPN tunnel.

Setting up the OpenSSL-based CA

Public-key cryptography by itself is susceptible to man-in-the-middle attacks because the identity of public-key owners cannot be easily established. When you retrieve a public key

(e.g., from the Web or from a received e-mail) there is a risk that an attacker that impersonates the peer you want to communicate with has replaced the peer's key with his own public key. The goal of a *public key infrastructure (PKI)* is to establish trust by binding public keys to identities.

Digital certificates are an essential component of a PKI. The certificate contains, among other things, a public key and a subject *distinguished name (DN)*. The DN is an identifier of the entity owning the public key carried by the certificate. The entity can be anything that requires cryptographic services such as a person, an organization, or a computer program. The *certificate issuer* signs the certificate using his private key. The certificate owner makes his certificate available so that others can communicate securely with him/her. There is nothing secret in the certificate and therefore any means of delivery can be used. However, the certificate recipients must control the certificate authenticity before using the enclosed public key. This is accomplished by obtaining the public key of the certificate issuer and use that to verify the certificate signature. The issuer's public key is made available also through a certificate. This certificate must either be verified by another issuer higher up in the chain or its authenticity be trusted for specific reasons.

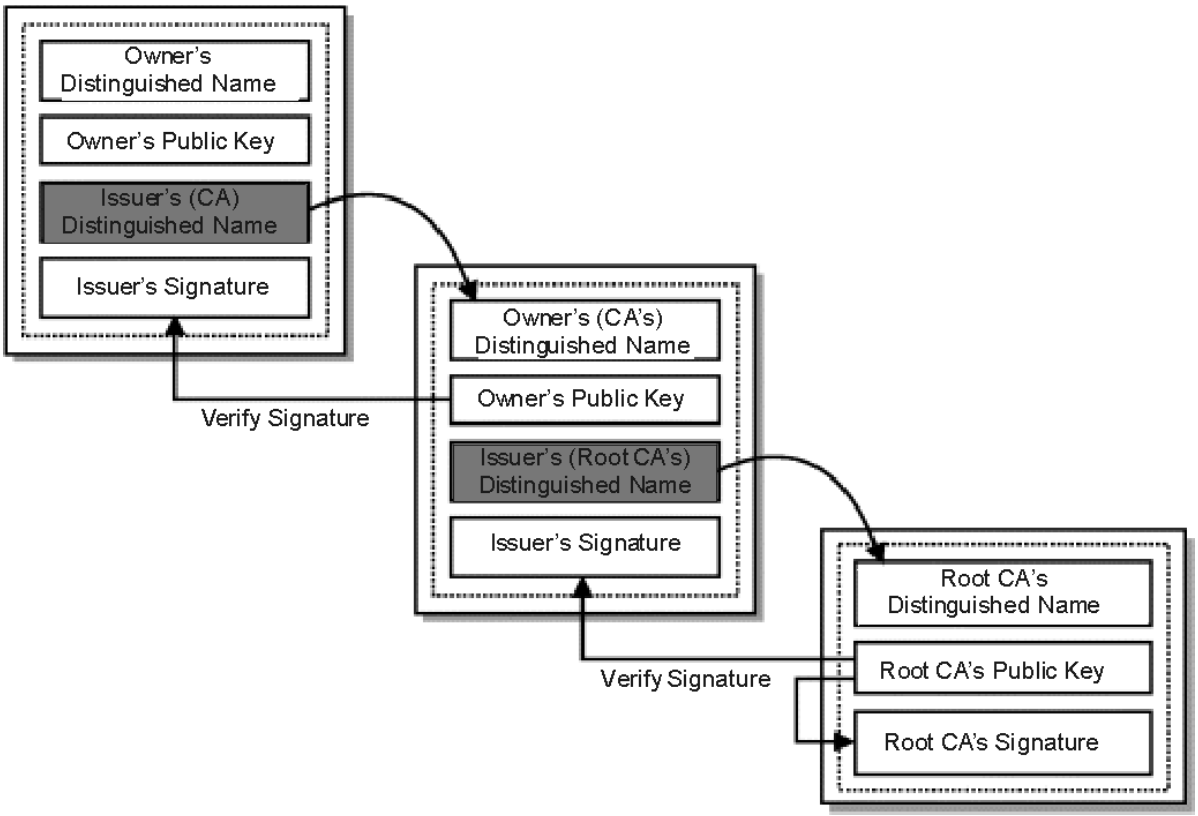


Figure 1 Verifying a certificate chain (note that root is at the bottom, in contrast to the layering presented above) http://security.stackexchange.com/questions/56389/ssl-certificate-framework-101-how-does-the-browser-actually-verify-the-validity

A certification authority (CA) is a person, or more likely an organization, that enjoys the trust of many users. Because of business or practicality reasons, CAs can organize themselves hierarchically. At the bottom are the end-users who ask an intermediate CA (e.g., a regional CA) to sign their certificates. The intermediate CA's certificates are signed by a CA from a higher level (e.g., a national CA). Eventually, the CA on the highest level is reached. This CA is referred to as the root CA and its certificate is called *root certificate*. Root certificates are

self-signed by the root CA since there is no CA on a higher level that can vouch for the identity of the root CA. A complete certificate verification implies that all certificates in the chain are verified, from the level below root to the end-user level. Failure to do so is a serious security risk. Root certificate cannot be verified the same way. Instead they are trusted by reputation and often included in products shipped to users (e.g., in web browsers or embedded devices such as smart phones, tablets, TVs).

There are two types of CAs: public and private.

Public CAs (e.g., Verisign) have built up trust with the public because they have a financial interest in providing certification services for people and organizations. Public knowledge that fraudulent certificates were issued (either by intent or mistake) has a catastrophic impact on the trust enjoyed by the CA and consequently on its profit coming from certification services.

Private CAs are responsible for issuing certificates within an organization, such as within an enterprise. In the case, the trust originates from the employer-employee relationship. A rational employer is very keen on maintaining safe access to digital resources. Consequently, it is in their best interest to implement adequate security procedures, such as managing digital certificates for the employees. Certificates from private CA can secure an intranet web site, enable an S/MIME e-mail system or allow clients to authenticate to the server. Since private CAs are not trusted outside their organization, their certificates cannot be used securely by others than organization members. **In this lab, you will setup a private CA.**

The *de-facto* standard format for digital certificates is X.509. The initial version of this standard was released by ITU in 1988. In 1995, IETF created the Public-Key Infrastructure x.509 Working Group (PKIX-WG) to standardize PKI usage over the Internet, which included guidelines on how to use X.509 (documented in RFC 4210[1]). The initial version of X.509 had numerous shortcomings and resulted in the release of subsequent versions, with X.509v3 being the latest version. PKIX-WG has document its use in RFC 5280[2].

The most notable change in X.509v3 is the support of extensions. They allow a certificate to contain additional fields beyond those defined in X.509v1. Some extensions were standardized in X.509v3 due their usefulness, but nonstandard extensions are possible as well. You can see the format for X.509 certificates in Figure 2.

An extension consists of a name (the name of the field), a value assigned to the field and a boolean flag indicating if the extension is critical or not. Applications that process a certificate with *critical extensions* must reject the validity of the certificate if any of the critical extensions are not recognized. Non-critical extensions that are not recognized by the application can be ignored.

[1] https://tools.ietf.org/html/rfc4210
[2] https://tools.ietf.org/html/rfc5280

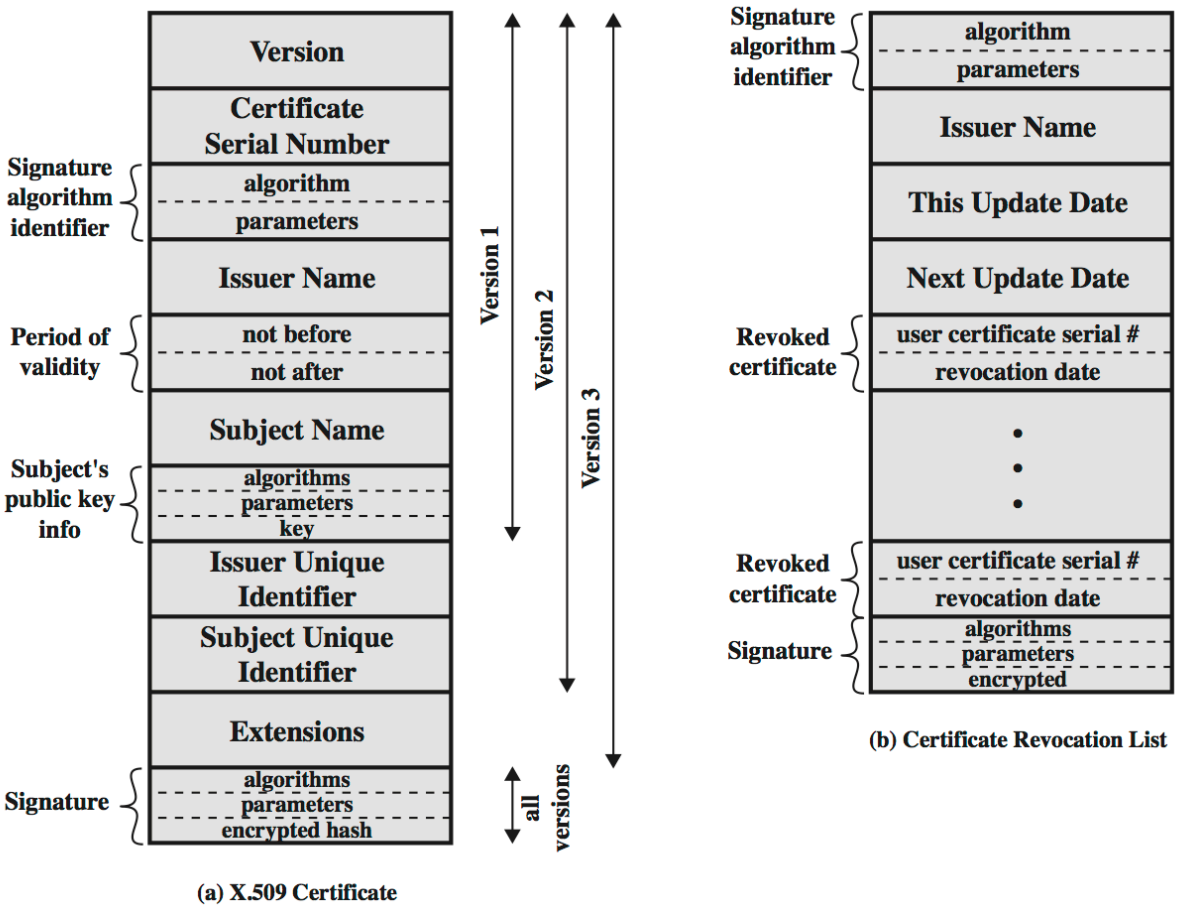


Figure 2 Format of X.509 certificate and CRL. From Stallings, "Cryptography and Network Security"

Guidelines

Best practice is to create a root CA and one or more intermediate CAs. The root CA's private key is kept offline, by a trusted (and often vested) member of the organization. This member can delegate the responsibility of issuing certificates to other members of the organization that will act as intermediate CAs. The trusted employee can also play the role of an intermediate CA, in addition to being root CA. Key-pairs and certificates are created for each of the intermediate CAs. The certificate for an intermediate CA is signed with the root CA private key.

If suspicion arouses that an intermediate CA had its private key compromised, then the root CA can revoke the intermediate CA certificate. This effectively destroys the certificate chain and indirectly invalidates all certificates signed by the compromised CA. The same revocation procedure can be applied when the individual acting intermediate CA leaves the organization. This will ensure that the person is unable to sign any more certificates on behalf of the organization.

Environment

Under Ubuntu Linux, by default, OpenSSL loads its configuration from */etc/ssl/openssl.cnf*. The default behavior can be changed by providing the name of the configuration file on the command-line and this is the approach used here. We begin by setting up a CA directory structure and then copy the default OpenSSL configuration. All our changes will be done to the copy, never to the original. This will ensure that other Linux application that rely on OpenSSL will continue to function even if you make errors in the configuration.

Create a main directory named <*acronym*>*_ca* under your home directory, where <*acronym*> is to be replaced by your BTH student acronym as it appears on the student portal. I will use the acronym *dril12* in the examples. First line in the code below changes current directory to your home directory. If current directory is your home directory, the command will have no effect.

```
cd
mkdir dril12_ca
```

Now we create the structure under the main directory.

```
cd dril12_ca
mkdir certs crl newcerts private
chmod 700 private
touch index.txt
echo 1000 > serial
pwd
```

On line 2 we create four sub-directories. When a certificate is issued by the root CA, it is placed automatically as a file under the *newcerts* directory. The filename is the serial number of the issued certificate. This is not always so practical – sometimes you want to look up a certificate by the owner's name for example. Therefore, we store under *certs* directory a copy of each certificate issued by the root CA. The *crl* directory will contain lists with revoked certificates. Finally, private keys are stored under the *private* directory. The private keys must be kept secret at all times. That is why we set the permission bits on the directory in line 3 to allow only the owner of the directory to access its contents.

OpenSSL will use the *index.txt* file to keep track of all issued certificates. Each certificate must have a unique value that is obtained from the serial file. OpenSSL increments the value in this file for each created certificate. On line 4, we provide the initial value. OpenSSL interprets the contents of the file to be a hexadecimal value, so in reality 1000 is actually 0x1000, which is 4096 in decimal. This also means that OpenSSL expects to find an even number of digits in the file.

Finally, line 5 prints the current working directory to the standard output. This will show something like */home/ats/dril12*, but using your acronym instead. The instructions below will refer to this as *the absolute (directory) path to your root CA directory*.

The next step is to copy */etc/ssl/openssl.cnf* to your CA directory structure.

cp /etc/ssl/openssl.cnf /home/ats/dril12_ca

Customize the OpenSSL configuration file

Open the *openssl.cnf* file from your CA directory with your favorite editor (e.g., leafpad or atom).

The file is divided into a number of sections, where the beginning of each section is identified by its name in square brackets (e.g., *[ca]*). The contents of a section consist of lines that are either a comment (identified by a hash # character at the beginning of the line) or parameter assignment. For parameter assignment, the part to the left of the equal sign is the parameter name, and the part to the right is the parameter value. It is possible to define own variables (parameters) as long as you are careful not to use OpenSSL built-in parameter names. It is difficult to make this distinction in the beginning, and therefore you should avoid creating own variables. If a parameter name with the dollar $ character prepended to it is placed on the right side of the equal sign, it will cause OpenSSL to replace it with actual parameter value (macro expansion).

Parameter values set in a section apply only within the scope of that section. Parameter values set outside any section have global scope (they are visible in all sections).

Find the *CA_default* section in the file and replace the *dir* parameter value with the absolute (directory) path to your root CA directory.

We will apply a particular naming scheme to all files (keys, certificates, CRLs) created with OpenSSL: <*name*>.<type>.<*format*>. The <*name*> must be replaced with the name of the owner which should be self-evident from the context (e.g., *root* for your root CA, *192.168.70.5* for a host with IP address 192.168.70.5). The <*type*> denotes the contents of the file: *cert* for a certificate, *cert-chain* for a certificate chain, *key* for a private key, *pub* for a public key, and *crl* for a CRL. We need therefore to change the entries in the *CA_default* section to reflect this scheme. You may also need to change the directory names to follow the structure described in the Environment section. In this lab, the <*format*> is always pem. Besides PEM, there is also a DER and a PKCS#12 format you should be aware of. DER is a binary format using ASN.1 encoding to store cryptographic objects in files. PEM contains the same information as DER, but binary content is base64 encoded (printable ASCII). PKCS#12 is a complex format that can be used store multiple types of cryptographic objects in the same file.

For example, the *certificate* and *private_key* parameter entries should be changed to:

certificate = $dir/certs/root.cert.pem
private_key = $dir/private/root.key.pem

In the same section, find the parameter *default_md* and make sure the value is set to *sha256*. The default value, *sha1*, is deprecated by Microsoft, Apple and Google for use as message digest (hash value). See https://www.schneier.com/blog/archives/2012/10/when_will_we_se.html for an explanation.

Also, comment out the *x509_extensions* parameter in this section. Extensions for a specific purpose are grouped together in a common section. We'll select the desired extension section from the command line.

We'll now move to the *req* section, which is used when creating certificates or certificate signing requests. Here, you will add the *default_md=sha256* parameter-value pair to ensure the SHA-2 digest is used.

In the next step, you edit the defaults under *req_distinguished_name* section. This section defines what elements are required to build a distinguished name (DN), which is a way to uniquely identify the certificate owner. The DN is built by concatenating several key-value pairs from the certificate (e.g., "CN=Mark Twain, OU=Authors, O=Independent C=UK"). The *Common Name (CN)* is usually the friendly name assigned to the entity owning the public key (e.g., the DNS name or IP address for a server). You don't need to change the CN here, but you will be prompted for it when you create the certificate.

countryName_default = SE
stateOrProvince_default = Blekinge
localityName_default = Karlskrona
0.organizationName_default = ET2540

The *v3_ca* section is used when creating a root certificate. Make sure the parameter *keyUsage* is set to "*critical, digitalSignature, cRLSign, keyCertSign"*. This parameter is a X.509v3 standardized extension field.

Task 1: [v3_ca]

Lookup the extensions enabled in the *v3_ca* section (all of them, not just the one mentioned above) in *man x509v3_config* and in RFC 5280 https://tools.ietf.org/html/rfc5280 and describe their purpose. What do the values assigned to these extensions mean?

We will now create an entirely new section called v*3_intermediate_ca*.

[v3_intermediate_ca]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true, pathlen:0
keyUsage = critical, digitalSignature, cRLSign, keyCertSign

Task 2: [v3_intermediate_ca]

Lookup the extensions enabled in the *v3_intermediate_ca* section in *man x509v3_config* and in RFC 5280 https://tools.ietf.org/html/rfc5280 and describe their purpose. What do the values assigned to these extensions mean? What is different compared to the *v3_ca* section?

The *usr_cert* section will be used for signing client certificates, such as those for e-mail and remote authentication. In that section, ensure that the following parameters (X.509v3 standard extensions) are set as shown below:

keyUsage = critical, nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = clientAuth, emailProtection

Task 3: [usr_cert]

Use the same information sources as in Task 1 and Task 2 to explain the purpose of the extensions enabled in *usr_cert* section (all of them, not just the ones mentioned above) and the effect of the values assigned to them.

We will create a *server_cert* section that will be used when signing server certificates, such as those used for web servers. You can place this section immediately after the *usr_cert* section

[server_cert]
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid,issuer:always
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth

Task 4: [server_cert]

Use the same information source as before to explain the purpose of the extensions enabled in *server_cert* and the effect of the values assigned to them.

We turn our attention now to the intermediate CAs. A single intermediate CA is need in this lab. We can call it CA1 and create a *ca1* directory for it under the root CA directory (*/home/ats/dril12/dril12_ca* in my case). Under *ca1* directory we create the same directory structure as found under the root CA directories. In addition, we create a *csr* directory to store all incoming certificate signature requests (CSRs). We'll explain what a CSR is when creating the intermediate certificate.

mkdir ca1
cd ca1
mkdir certs crl newcerts private csr
chmod 700 private
touch index.txt
echo 2000 > serial
echo 2000 > crlnumber
cp ../openssl.cnf .

The penultimate command creates the file *crlnumber* that is used to keep track of CRLs. The purpose of the last command above is to copy the OpenSSL configuration file from the root CA directory to the intermediate CA directory. This copy needs to be customized for CA1.

First you need to change the parameter entries under *[CA_default]* to match the new directory, *ca1*, as well as the <*name*> of the new entity, *ca1* (so that the name of files produced with OpenSSL for the intermediate CA start with *ca1*, not *root*). This is like the changes you implemented above.

Secondly, the value for the parameter *x509_extensions* in the same section must be set to *usr_cert* and the value of the parameter *policy* must be set to *policy_anything*.

Task 5: Policies

Compare the *policy_match* section with the *policy_anything* section. What do they do, what are the differences?

We have now completed the OpenSSL configuration required by the root CA and intermediate CA and can proceed to the next section that deals with the creation of necessary cryptographic material to operate *root* and *ca1*.

Keys and certs for root and ca1

The next step in getting our PKI operational is to generate a key-pair and corresponding certificate for *root* and *ca1*, respectively. The process is slightly different for these two, because *root* is a root CA that self-signs its certificate whereas *ca1* has its certificate signed by *root*.

We will use RSA as our public-key encryption algorithm. There is no specific reason for choosing this algorithm other than very good support for it in the software used for this lab. *Root* and *ca1* will use RSA keys with length 4096 bits. For added security, the generated keys will be encrypted with the AES symmetric encryption algorithm using a 256-bit key derived from a chosen password.

The comments shown below assume that they are execute from the root CA directory (*/home/ats/dril12* in my case). Enter the following commands[3] to create the private RSA key for *root* and *ca1*, respectively:

openssl genrsa -aes256 -out private/root.key.pem 4096
openssl genrsa -aes256 -out ca1/private/ca1.key.pem 4096

For each generated RSA key, you will be asked to enter a password. The password is used to derive the AES-256 key that will encrypt the RSA key. **Use the same password as used to**

[3] OpenSSL is huge library of cryptographic functions. The first command-line argument after *openssl* denotes which function you are calling. See *man openssl* for a complete list of available functions and *man genrsa* for documentation for the function used to generate an RSA private key.

log into the VMs. Note also the path to the generated key (after the *–out* option), and the key names. They should match the changes you made to the *openssl.cnf* files for *root* and *ca1*.

Another thing you should remark is that we have generated only private keys. However, the RSA private keys contain enough information to generate the public key as well[4]. The internal syntax of the RSA private according to RFC 8017[5] is as follows:

```
RSAPrivateKey ::= SEQUENCE {
        version             Version,
        modulus             INTEGER,  -- n
        publicExponent      INTEGER,  -- e
        privateExponent     INTEGER,  -- d
        prime1              INTEGER,  -- p
        prime2              INTEGER,  -- q
        exponent1           INTEGER,  -- d mod (p-1)
        exponent2           INTEGER,  -- d mod (q-1)
        coefficient         INTEGER,  -- (inverse of q) mod p
        otherPrimeInfos     OtherPrimeInfos OPTIONAL
}
```

You should recognize the notation, since we used the same variable names in the lectures. To generate the public key, you need only the modulus and public exponent from the private key. Indeed, per RFC 8017

```
RSAPublicKey ::= SEQUENCE {
        modulus             INTEGER,  -- n
        publicExponent      INTEGER   -- e
}
```

To obtain the public key (e.g., for root.ca.pem) using OpenSSH you can use the following command[6]:

openssl rsa -in private/root.key.pem -pubout -out root.pub.pem

We will not use public keys directly. They will be used indirectly after each being embedded in a certificate.

For additional protection, the filesystem access rights to private keys should be restricted to the owner:

chmod 400 private/root.key.pem
chmod 400 ca1/private/ca1.key.pem

---

[4] When we talked about RSA key generation, we said that a key-pair is produced. The two keys from the pair are equivalent in functionality and it is up to the user to decide which one is public and which one is private. OpenSSL creates a key that is marked for private use, and therefore makes this choice on behalf of the user.

[5] https://tools.ietf.org/html/rfc8017

[6] See *man rsa*.

The next step is to generate the self-signed certificate for *root*. Whereas key generation did not require OpenSSL configuration information, the certificate generation depends on it. It is crucial that the correct *openssl.cnf* file (for *root* or *ca1*) is specified on the command-line. Failure to do so will cause the *openssl* to exit with an error or generate incompatible cryptographic objects.

To generate the self-signed certificate entering the following command[7]:

openssl req -config openssl.cnf -key private/root.key.pem
-new -x509 -days 7300 -sha256 -extensions v3_ca
-out certs/root.cert.pem

First, you will be prompted for AES-256 password protecting the root private key. Then, you will be prompted to enter values for several parameters. Press enter to accept the defaults. When prompted for Common Name (CN), enter your name followed by the word Root. This will help you later when viewing the certificate to recognize it as your root certificate.

Task 6: Options for the root certificate

Use *man req* to lookup the options passed to the command above and explain what they and the assigned values do.

Task 7: Verify the root certificate

Use the command below to verify the root certificate. Include the output in your report.

openssl x509 -noout -text -in certs/root.cert.pem

At this stage, we will create the intermediate CA certificate. This is no longer a self-signed certificate – it will be signed by the root CA. It is therefore a two-step process.

In the first step, we create a certificate signing request (CSR) on behalf of CA1. The CSR contains (among other things) CA1's public key and an identifier for a digital signature algorithm. The CSR will be signed with CA1's private key using the specified digital signature algorithm.

During the second step, our root CA signs the CSR with its own private key. The signed CSR is the certificate for *ca1*.

The reason for splitting the process in two steps stems from the mode of operation of public CAs. In that scenario, end-users generate the private key and under no circumstances will share it with the public CA, like we do here. In a private CA scenario, the organization may want control over the private keys assigned to users (e.g., to maintain ability to decrypt documents after a user leaves the organization).

It is important to specify the OpenSSL configuration file for CA1 when creating the CSR, so that the correct filename path and policies are used. **Very important is also, when**

[7] The command is split on multiple lines to enhance the presentation. Everything must be typed on the same line. For details on certificate generation, see *man req*.

prompted, to specify a different Common Name (CN) than the CN used for the root certificate. Enter your name followed by the word CA1 when prompted for the CN. This way you will be able to recognize this as CA1's certificate. Enter the command below to create the CSR:

openssl req -config ca1/openssl.cnf -new -sha256 -key ca1/private/ca1.key.pem
-out ca1/csr/ca1.csr.pem

The first prompt will request you to enter the AES-256 password protecting the CA1 private key.

Task 8: Verify the CSR

Enter the command below to verify the CSR. Include the output in your report.

openssl req -text -noout -verify -in ca1/csr/ca1.csr.pem

We are now ready create the certificate for CA1 using the CSR. Since the root CA is signing the certificate it is important to use *root's* OpenSSL configuration file. Enter the following command:

openssl ca -config openssl.cnf -extensions v3_intermediate_ca -days 3650 -notext
-md sha256 -in ca1/csr/ca1.csr.pem -out ca1/certs/ca1.cert.pem

You will be prompted to enter the AES-256 password protecting the root private key.

Task 9: Options for intermediate CA certificate

Use *man ca* to lookup the options passed to the command above and explain what they and the assigned values do. In particular, explain what is the effect of specifying the *v3_intermediate_ca* value for the *-extensions* option?

Task 10: Verify the certificate for CA1

Enter the commands below to verify the certificate for CA1. Include the output in your report. First command prints the cert and the second verifies the CA1's certificate authenticity against the root certificate.

openssl x509 -noout -text -in ca1/certs/ca1.cert.pem
openssl verify -CAfile certs/root.cert.pem ca1/certs/ca1.cert.pem

---

At this point you are ready to sign end-user (client, server) certificates. However, when those certificates are presented to a 3rd party application, that application will want to verify the entire certificate chain (root CA certificate, intermediate CA certificate and end-user certificate). The application obtains the end-user certificate through natural interaction with the certified entity (e.g., a web server provides its certificate to a web server when they interact to setup an HTTPS connection). However, it is not the entity's responsibility to

provide to the application the root and intermediate CA certificates as well. Instead, the application can either retrieved them dynamically (based on URLs found in the end-user certificate) or have them installed manually by the application user. The second approach is often used by application manufacturers to install root certificates before shipping to customers.

We will use the second approach here. For convenience, we will create a certificate chain file containing both intermediate CA and the root CA certificate, respectively. This is as simple as concatenating the two files. Execute the commands below to create a certificate chain:

cat ca1/certs/ca1.cert.pem certs/root.cert.pem > ca1/certs/ca1.cert-chain.pem
chmod 444 ca1/certs/ca.cert-chain.pem

The last command ensure that the certificate chain is accessible to all users on Server A.

Create and sign a server certificate

If you have completed the 10 tasks above, you have learnt everything you need to know for issuing certificates. I'll just summarize the necessary steps and then in Task 11 you'll create a server certificate on your own.

1. Create an RSA private key for the server
2. Generate a CSR using the RSA private key from the previous step
3. Use CA1's private key to sign the CSR and create a certificate for your server. You must include the *server_cert* extensions in the certificate.

There is one little twist related step 1. Previously, when we created RSA private keys, we have instructed OpenSSL to encode the generated key with AES-256. AES-256 requires a secret key, which was derived from the password we entered in the terminal. It is fully possible (and in many cases even recommended) to use the same procedure for the server certificate. However, you need to understand the implications.

Every time the server is restarted it must decrypt the private key. The server does not have the password used for decryption, so it must somehow prompt the user/admin to enter it. This means that if the server restarts automatically (e.g., because of a power loss or if configured to automatically install security patches), it will not become operational without operator intervention. In some cases, the server may be able to obtain the password from a configuration file. In that case, you must make sure the file is well protected by setting the appropriate permission bits. This solution is less secure because in the event of a security breach an attacker may be able to read your password. The security impact is even higher if you reuse the password for multiple purposes.

A third option is to skip the encryption of the private key altogether. Obviously, the key must be protected by setting very restrictive permission bits. Nonetheless, this is probably the least secure solution. If an attacker circumvents filesystem protection, he or she will be able to copy the private key and potentially impersonate the key owner and launch man-in-the-middle attacks. This is also the most comfortable option and this is what you will use in the lab, despite the security implications. To avoid encryption of the private key, skip the *-aes256* option when generating the key.

Task 11: Create server certificate

Create a server certificate using RSA 2048 bits (weaker encryption than before). Skip encryption of the generated key. The certificate must be signed with CA1's private key. Enter *localhost* when prompted for Common Name (CN).

For convenience, place each of the generated files **under the corresponding directory** in the CA1's directory structure.

Use *openssl* to dump the contents of the CSR and the signed certificate, respectively, and include the output in the report. Verify the server certificate against your certificate chain and explain how you did it.

Configure Apache web server to use the server certificate

You will now configure the Apache web server on Server A to use the certificate created in Task 11. To do that, you must copy the private key and the server certificate in the folders used by Apache to search for cryptographic material.

The server's private key must be copied to the folder */etc/ssl/private/* and the certificate under */etc/ssl/certs/*. Apache needs also access to the certificate chain (to verify it is not using a bogus cert). The certificate chain for CA1 must be copied to the folder */etc/apache2/ssl.crt/*.

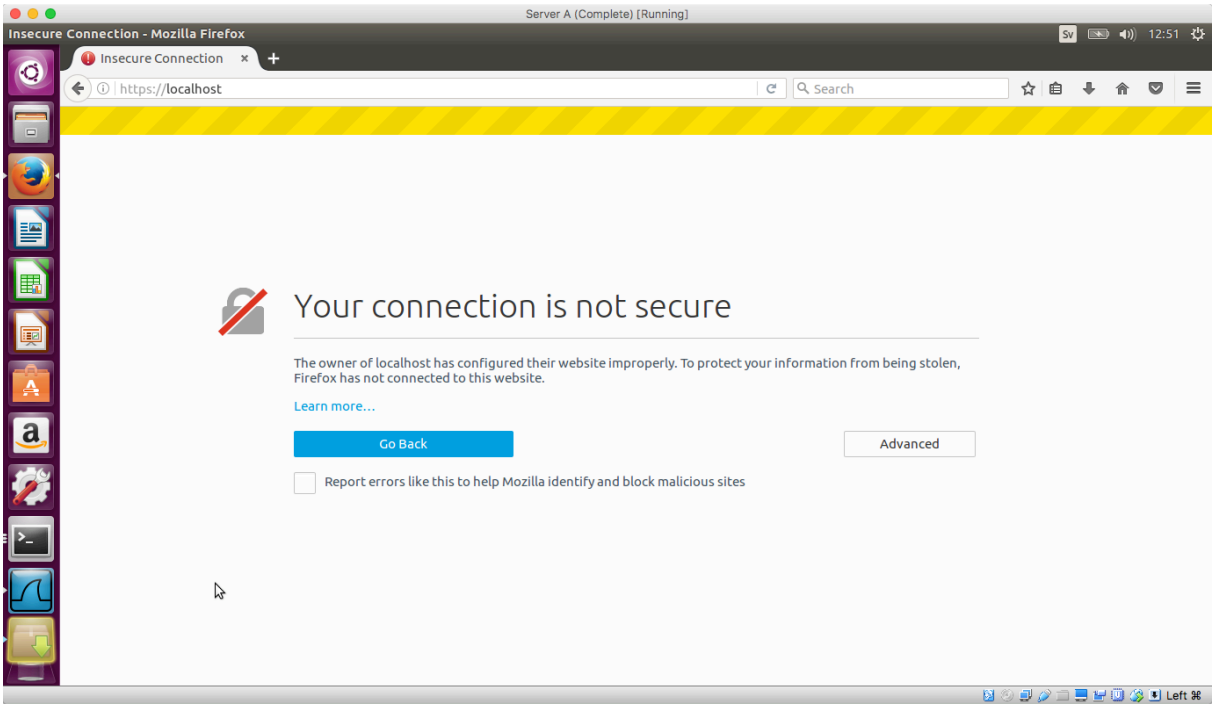
Apache's configuration must be modified to use your cryptographic material. Edit the file */etc/apache2/sites-enabled/default-ssl.conf* and enable the parameters *SSLCertificateFile*, *SSLCertificateKeyFile* and *SSLCertificateChainFile*. Make sure the parameter values match your files.

After saving your changes you must to restart Apache to apply the changes:

sudo service apache2 restart

Add certificate chain to Firefox web browser

Start the Firefox browser on Server A and enter https://localhost in the URL field. Firefox should display a warning as shown below:

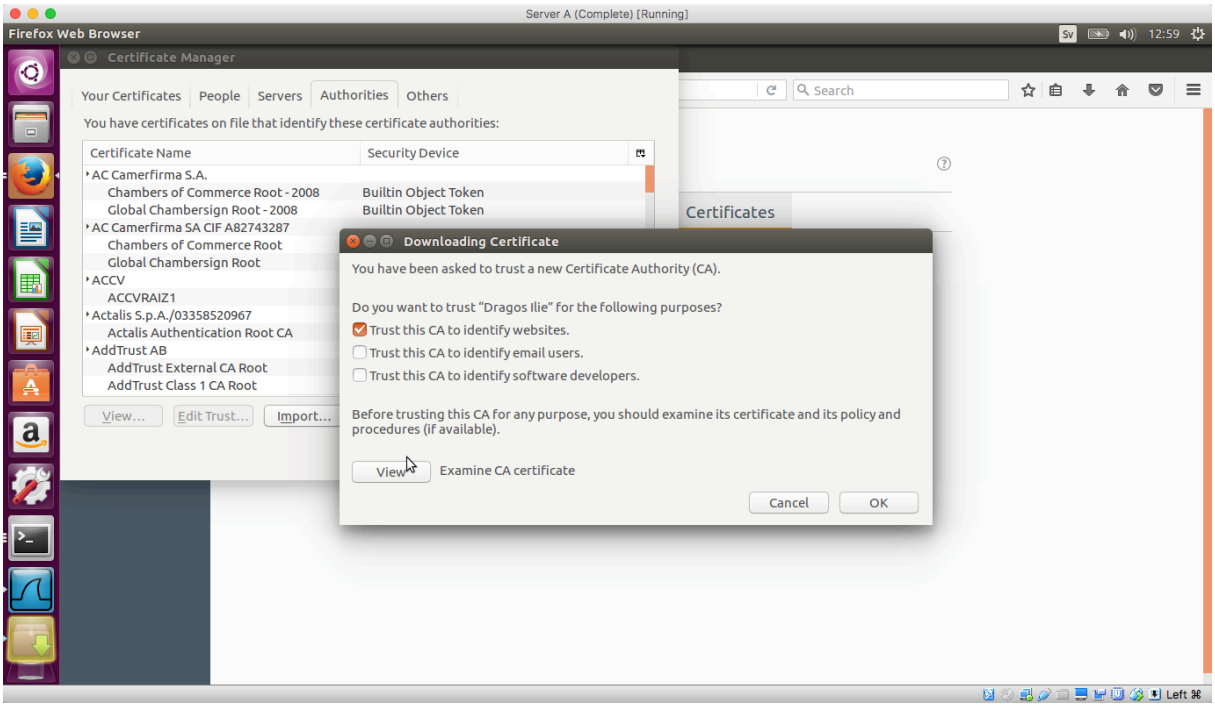


If you don't see the warning try reloading the page while keeping Shift press to force the page to be retrieved from server instead of being served from cache.

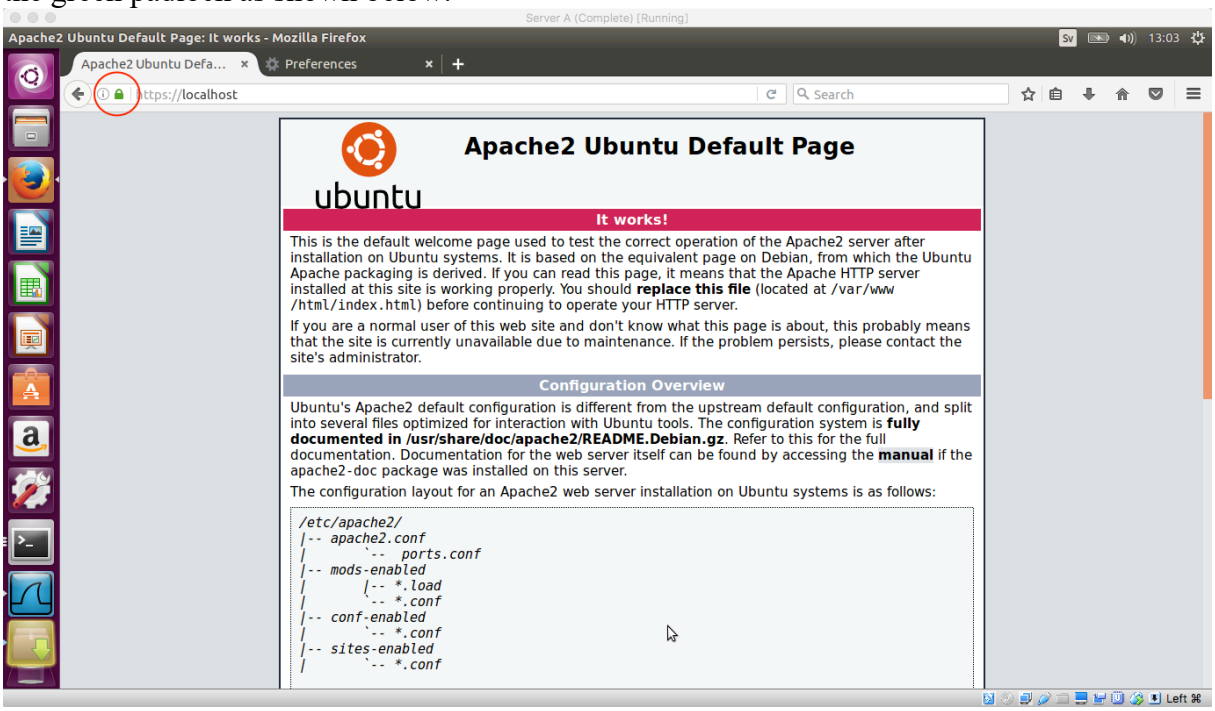
The reason you get the warning is because Firefox cannot verify the entire certificate chain because it does not have access to your root CA and intermediate CA certificates.

Select in the Firefox menu Edit->Preferences. Then, select Advanced on the left pane and Certificates in the main window. Click View Certificates.

Go to Authorities, click Import and select your certificate chain. You should a screen similar to the one shown below:

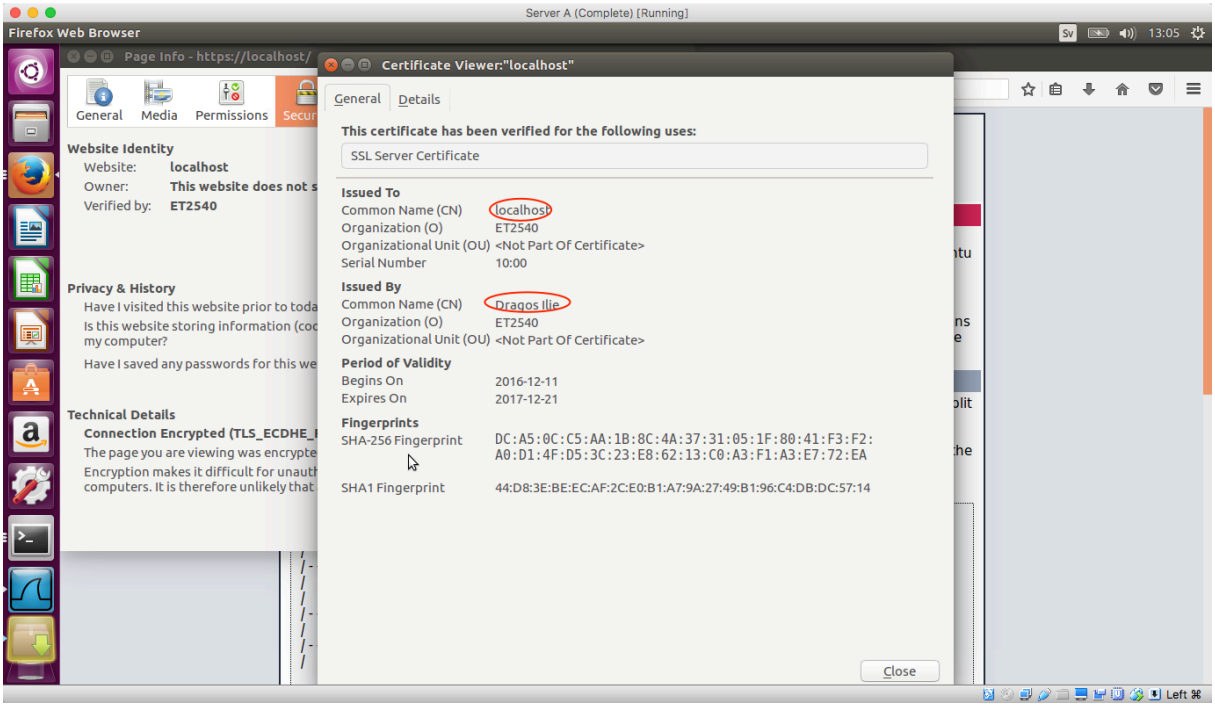


Reload the page https://localhost while pressing Shift to avoid a cached version. You should the green padlock as shown below.



Task 12: Show your certificate in Firefox

Click on the green padlock and retrieve the certificate details. Include a snapshot of it in the report, as shown below.



Revoke the certificate

A certificate is valid during a certain validity period decided when the certificate is issued. Certificates that have exceeded the validity period are invalid and must not be used. In some cases, it is necessary to revoke a certificate before it expires. Valid revocation reasons are compromise of owner's private key, a user no longer being certified by a CA (e.g., an employee leaving a company), or the compromise of CA's certificate (e.g., the CA private key is leaked, or CA infrastructure is hacked and fraudulent certificates are issued).

Revoked certificates are maintained in a certification revocation list (CRL). Entities that verify certificates need to obtain CRLs from CAs. One way to support CRL distribution is provided by the X.509v3 extension called CRL Distribution Points[8]. This is a sequence of Distribution Points, where each Distribution Point contains at least one mechanism to retrieve the CRL. Usually, this an HTTPS-based URL.

The idea is that an application (e.g., a web browser) verifying a certificate also retrieves the CRLs form the CRL Distribution Points. Unfortunately, most popular browsers are moving away to a different distribution mechanism called Online Certificate Status Protocol (OCSP). OCSP tries to solve three problems related to CRL Distribution Points. First one, is that the set of CRLs maintained by a CA can grow quite large over time. This has a strong impact on the latency in setting up a secure connection. Second problem is that CRLs are usually cached by the application as a measure to handle the latency problem. Depending on the cache policy, there is a lag time between the moment an entry is added to the CRL until the time the application receives the latest change. Lastly, CA don't necessarily generate a new CRL every time a certificate is revoked. Rather, they generate the CRLs periodically. This again creates a lag time until the application detects a revoked certificate.

Setting up OCSP is outside the scope of this lab. You will instead use the traditional method of creating CRL Distribution Points. This is not without a merit – strongSwan, the IPsec VPN system you will use in the remainder of the lab, supports CRL Distribution Points.

To enable CRL Distribution Lists extension you need add the *crlDistributionPoint* parameter to the list of extensions used when creating a certificate. In our case, this is needed only for server certificates.

Edit the CA1 OpenSSL configuration file and add the following parameter to the *serv_cert* section:

crlDistributionPoints = URI:https://localhost/ca1.crl.pem

[8] See https://tools.ietf.org/html/rfc5280 and *man x509v3_config*.

Task 13: Create a CRL for CA1

Enter the following command to create a CRL for CA1:

openssl ca -config ca1/openssl.cnf -gencrl -out ca1/crl/ca1.crl.pem

Dump the contents of the CRL using the following command and include the contents in the report:

openssl crl -in ca1/crl/ca1.crl.pem -noout –text

You are now ready to revoke a certificate. In the next task, you will create a fake certificate and then you will revoke it.

Task 14: Revoke a certificate

Create a **user certificate** (*attention*: you will need to user the *usr_cert* extensions) using RSA 2048 bits (weaker encryption than before) for entity dragos.ilie@bth.se. Skip encryption of the generated key. The certificate must be signed with CA1's private key. Enter *dragos.ilie@bth.se* when prompted for Common Name (CN).

For convenience, place each of the generated files **under the corresponding directory** in the root CA's directory structure.

Use *openssl* to dump the contents of the CSR and the signed certificate, respectively, and include the output in the report. Verify the server certificate against your certificate chain and explain how you did it.

Now, revoke the newly created certificate by entering the following command:

openssl ca -config ca1/openssl.cnf –revoke ca1/certs/dragos.ilie@bth.se.cert.pem

Copy the contents of the file *ca1/index.txt file* to the report. It should show that the certificate is revoke (letter R in the being of the entry).

Recreate the CRL as was done in Task 13, dump its contents and include them in the report. The revoked certificate should appear there.

---

strongSwan IPsec VPN

At this point you will practice setting up an IPsec VPN. You will use strongSwan, which is a complete OpenSource IPsec VPN solution that runs on Linux, Windows and Mac OS X. In addition, it is compatible with the IPsec features supported by Android- and iOS-based products.

It is important to understand that strongSwan does not encrypt the data exchanged over the VPN. The encryption is performed by the underlying operating system where strongSwan is running (Linux in your case). strongSwan is responsible for helping peers negotiate keys for

symmetric encryption using the Internet Key Exchange (IKE) protocol. strongSwan supports both IKEv1 and IKEv2. IKE allows peers to agree on the cryptographic parameters to use for encryption and then to mutually agree on an encryption key through the Diffie-Hellman key exchange algorithm.

IKEv1 was released in 1998 and eventually received criticisms related to vagueness and inconsistencies in the protocol specifications that could lead to implementations susceptible to various security attacks. Seven years later the specification for IKEv2 was released, where a somewhat simplified and more reliable version of protocol was presented. IKEv2 also has better support for features such as NAT traversal, Extensible Authentication Protocol (EAP), and mobility and multi-homing (MOBIKE).

strongSwan has implemented IKEv1 in a service called *pluto*, whereas the IKEv2 implementation is provided by a service called *charon*. You will use IKEv2 (charon) for this lab.

In addition to key negotiation, strongSwan provides a more user-friendly way to configure the operating system to encrypt/decrypt IP traffic. It is possible to configure the encryption manually, but is a very tedious process. Also, features such as dynamic keying (changing the encryption key frequently) are not suitable for a manual approach.

Linux IPsec support

There are multiple competing IPsec stacks in Linux and one should look at the history to understand how they relate to each other. The KAME project started in 1998 was the first effort to implement a free IPsec stack that worked both for IPv4 and IPv6. The implementation was intended for BSD UNIX systems, but was soon available as a patch to the Linux kernel. At about the same time, John Gilmore (one of the founders of the Electronic Frontier Foundation known, among other things, for cracking DES encryption) started the project FreeS/WAN that aimed to bring IPsec to Linux. This also required patching the Linux kernel. The FreeS/WAN patch was referred to as Kernel IP Security (KLIPS).

In both cases (KAME and KLIPS), the patch was needed to allow the IKE service, which is running in user-space, to communicate with a counterpart inside kernel-space to install encryption keys and configure various encryption parameters.

IETF has defined the PF_KEYv2 in RFC 2367 as a standard API that allows user-space processes to configure IPsec in the operating system kernel. Both KAME and KLIPS made use of PF_KEYv2. Maintaining patches for both KAME and KLIPS for each kernel release was challenging and therefore a decision was made to make KAME the default IPsec stack in the upcoming Linux 2.6 kernel line. The KAME implementation under kernel 2.6 is referred to as NETKEY. However, the 2.6 kernel was featuring a revised protocol stack and one of the new features was a protocol called NETLINK. Unlike PF_KEYv2, NETLINK is a generic protocol that allows user-space entities to exchange messages with various parts of the kernel. For instance, RT_NETLINK is a NETLINK extension that enables a dynamic routing process to install or remove routes in the kernel. In a similar vein, XFRM_NETLINK[9] allows IKE or other user processes to manage the IPsec Security Association Database (SAD) and the Security Policy Database (SPD) maintained in the kernel.

[9] XFRM appears to be an acronym for “transform configuration”.

Nowadays, Linux supports natively both NETKEY/PF_KEYv2 as well as NETKEY/XFRM. The *iproute2* package (containing the *ip* commands that you encountered in the previous lab) makes use of NETKEY/XFRM, which is considered more powerful than the NETKEY/PF_KEY2.

All *iproute2* command begin with *ip*. You have already used the commands *ip addr* and *ip -4 route* in the previous lab to list assigned IP addresses and to show the IPv4 routing table, respectively. Later, in this lab, we'll introduce the commands *ip xfrm state* and *ip xfrm policy*, that are used to manage the SAD and SPD, respectively.

Linux IPsec packet handling

IPsec packet handling is neatly integrated with both the routing system as well as the Netfilter framework (the module that implements the *iptables* firewall).

You may recall from our firewall lecture that there are five default chains in Netfilter: PREROUTING, INPUT, FORWARD, OUTPUT, and POSTROUTING (see Figure 3 on next page). There are also four tables[10]: raw, mangle, nat, and filter. Each chain is associated with several tables, but only the OUTPUT chain uses all of tables.

Packets move from chain to chain. When a packet reaches a chain, it will traverse all the tables associated with the chain in the order listed above. You use the *iptables* command to insert rules in tables that define how packets are being handled in a particular chain. When you write, for example,

iptables -t filter –A INPUT <filter> -j <action>

it means that packets that reach the filter table in the INPUT chain and match the filter <filter> (e.g., src 10.0.0.5) will be subjected to <action> (e.g., DROP or ACCEPT).

The left side of Figure 3 shows the processing path for a packet arriving at a host's network card. The packet first enter the PREROUTING chain. Without going too much into details, inside the PREROUTING chain the packet will enter the raw, mangle and nat tables, in this order. The green box labeled Connection tracking refers to the kernel module that is used for implementing a stateful packet filter firewall. You manipulate this indirectly with *-m conntrack* option for *iptables* (you did this in the previous lab to track the state of TCP connections).

[10] There can be more than four tables if specific kernel extensions are turned on (e.g., SElinux)

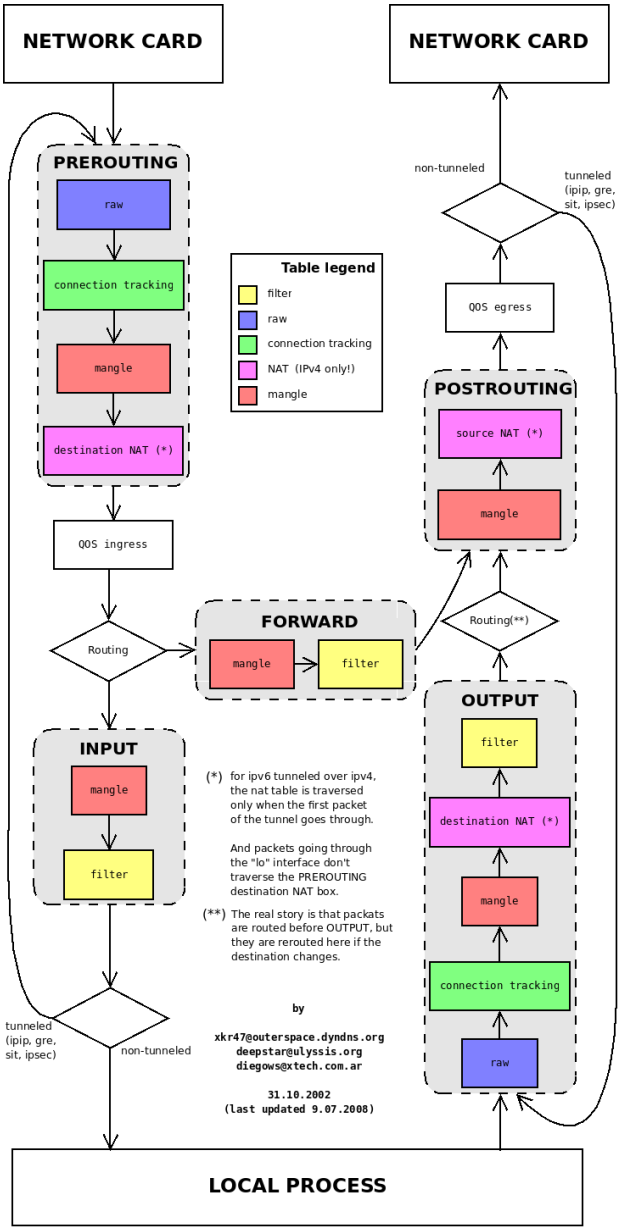


Figure 3 Netfilter packet processing (from Jonas Berlin aka. xkr47)
http://xkr47.outerspace.dyndns.org/netfilter/packet_flow/

If the packet is not dropped it is passed to the routing module. The module decides if the final destination of the packet is on the current host or if the packet must be forwarded to another host on the path to the final destination.

If this host is the destination host, the packet enters the INPUT chain and is passed to the mangle table and then to the filter table. ESP and AH packets[11] that are not dropped in the INPUT chain are passed to the IPsec module. The module will search the SAD for a security association (SA) that matches the packet. This is required to learn the correct cryptographic parameters for decoding/decrypting the packet. If none is found, the packet is dropped.

[11] As you should recall AH and ESP are the IPsec protocols used for authenticated and encrypted data. AH provides pure authentication services, while ESP can provide both authentication and encryption.

Otherwise, the packet is decapsulated (IPsed header removed), and decrypted if necessary, and the resulting packet is sent back to the PREROUTING chain where it is processed as any other regular packet.

A packet send by a local process on this host enters first the OUTPUT chain and traverses the raw, mangle, nat, and filter tables. Accepted packets are processed by the routing module where the outgoing network interface is selected. Then, the packet enters the POSTROUTING chain where, for example, NAT processing may occur. The kernel will check inside the SPD if a IPsec security policy matches the outgoing packet. If there is a match, the kernel will verify if a corresponding SA exists in the SAD. If none is found, the kernel will use the Netlink protocol to instruct IKE to establish a new SA. If an SA exists or can be established the kernel does the necessary IPsec processing (AH/EPSP) and the resulting packet is sent back to the output chain where it is processed as a regular packet.

Policy routing

strongSwan uses a networking feature called *policy routing*. This is different from the SPD and the security policies used by IPsec. Normal IP routing is based on the destination IP address field found in the IP header. A host with IP forwarding enabled will select the same path for all packets going to a specific destination. This situation is not always desirable. An early case for policy routing was support for different QoS levels for various customer classes. Gold customers were paying to receive high throughput and low latency service, while bronze customers settled for cheaper best effort services. Selecting routes for these customer involved, among other things, looking at the source IP address as well as destination address as part of the IP routing process.

The Linux policy routing implementation requires multiple routing tables and a routing policy database (RPDB). You can list the policies in the RPDB by entering the following command[12]:

ip rule show

The output should show something like:

```
0:      from all lookup local
220:    from all lookup 220
32766:  from all lookup main
32767:  from all lookup default
```

The leftmost column shows the priority assigned to the policy routing rule on the same line. Available rules are scanned from the highest priority (0) to lowest priority (32767) and the first rule that matches selects the routing table that will be used by the routing process to lookup the route. Once an RPDB policy matches, the indicated table is used to lookup the next hop. If the routing process is unable to compute the next hop for the packet from the routing table indicated by rule, processing continues to the next rule in RPDB. The “from” keyword denote a selector on the source IP address, but since the word “all” is used also, it indicates that all packets will match. The selectors can be much more complex, but we will not explore this any further. By default, strongSwan inserts a policy rule with priority 220.

[12] See *man ip-rule*.

When Ubuntu Linux starts, it sets up three tables: local (id 255), main (254), and default (253)[13]. The local routing table contains high priority control routes for local and broadcast addresses. This table is for internal use of the kernel and should never be modified from user-space. The main routing table is the normal table used when a table is not specified. Finally, the default table is usually empty and used when no other table matches.

To list the available routes in a table (e.g., local) use one of the following commands:

ip route list table local

or

ip route list table 255

The table with id 220 is created by strongSwan. It has no user-friendly name like the three tables above. The id number was selected to match the priority of the corresponding rule in RPDB, to ease recognizing the connection between the policy rule and the table. All routes create by strongSwan are inserted into table 220.

The use of policy routing does not change the IPsec packet handling presented in the previous section. The routing module consults the RPDB for every incoming or outgoing packet. Packets from or to remote destinations will not match any routes in the local table. Table 220 is the next table consulted (according to priority) and that will contain all strongSwan routes that trigger IPsec processing. If none of the entries in table 220 matches the destination, the routing module consults the main table. Currently, the RPDB selector for table 220 and the main table are identical, and therefore locating IPsec routes in table 220 is just a matter of not mixing them with the others (a matter of convenience). However, one could restrict the IPsec service to a narrower class of users (e.g., gold customers) by modifying the RPDB selector to match on specific IP blocks in the source IP address of the packets. All other customers would get the no-IPsec service from the default table.

Installing and configuring strongSwan

You will begin by installing strongSwan on Server A and on Server B. This is accomplished by entering the following command on each host:

sudo apt-get install strongswan

It is suggested that you reboot the VMs after installation to ensure all services are started.

The main configuration files for strongSwan are */etc/ipsec.conf* and */etc/ipsec.secrets*. You can find a large number of configuration examples at https://wiki.strongswan.org/projects/strongswan/wiki/IKEv2Examples.

It is **strongly** recommended that you begin by reading the strongSwan introduction at https://wiki.strongswan.org/projects/strongswan/wiki/IntroductionTostrongSwan.

[13] When using *ip route* on a particular routing, you can specify either the table id or the table name.

All VPN configuration options are documented in https://wiki.strongswan.org/projects/strongswan/wiki/IpsecConf, in particular the connections settings available at https://wiki.strongswan.org/projects/strongswan/wiki/ConnSection.

IKEv2 will always require some sort of a shared secret between site A and site B to setup a VPN, or private keys to negotiate a shared secret with the Diffie-Hellman algorithm. This type of information is stored in */etc/ipsec.secrets* and is documented in https://wiki.strongswan.org/projects/strongswan/wiki/IpsecSecrets.

strongSwan provides a wrapper around SSL that is intended to simplify the creation and maintenance of digital certificates. This is available in the form of *ipsec pki* commands. You are not allowed to use these commands. Instead, you are required to use the *openssl* command described in the first part of the lab. However, feel free to use the instruction on how to install certificates in strongSwan available at https://wiki.strongswan.org/projects/strongswan/wiki/SimpleCA.

It is a good idea to turn on detailed logging in the beginning. This can be done at runtime by issuing the *sudo ipsec stroke loglevel* command as explained on https://wiki.strongswan.org/projects/strongswan/wiki/Loggerconfiguration. You should enable level 4 logging for ike, chd and esp subsystems.

Environment

As you remember from the previous lab, server A and client A have each a host-only interface attached to the 192.168.60.0/24 network. The interface on server A is assigned the IP address 19.168.60.100 and the interface on client A is assigned the IP address 192.168.60.111. To make this resemble a realistic example, we'll say that server A and client A are part of site A, which "owns" the network 192.168.60.0/24. Server A provides Internet access to site A through its NAT interface.

Similarly, server B and client B are part of site B, which "owns" the network 192.168.80.0/24. The network interface on server B is assigned IP address 192.168.80.100 and the interface on client B is assigned IP address 192.168.80.111. Server B provides Internet access to site B through its NAT interface.

Both server A and server B have a secondary host-only interface connected to the network 192.168.70.0/24. Server A's interface is assigned IP address 192.168.70.5, while server B's IP address is 192.168.70.6. You will configure server A and server B to establish the VPN service across the 192.168.70.0/24 network. This corresponds to a scenario where the network 192.168.70.0/24 was provisioned for high-speed access, unlike the best-effort Internet service available through the NAT interfaces. A more practical reason for this setup is that establishing the VPN over the NAT interfaces requires handling multiple NAT layers, which makes this lab more difficult.

The 192.168.60.0/24 and 192.168.80.0/24 are *private networks* that will connected virtually over the "*public" network* 192.168.70.0/24.

Host-to-host transport mode VPN with PSK authentication

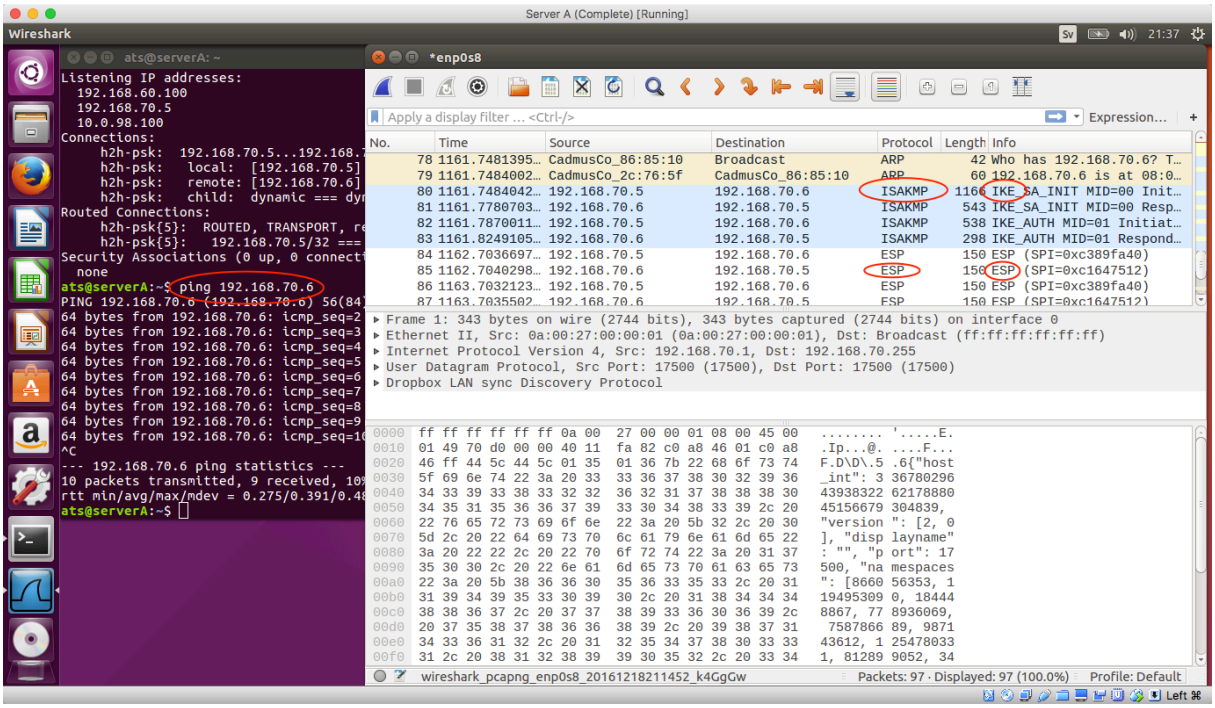
The first step is to establish a simple host-to-host transport mode VPN between Server A and Server B. The peers will authenticate using a pre-shared key (PSK), which is a plain-text

password configured in */etc/ipsec.secrets*. Use the same password as before in the lab. Also, it is suggested that you set *auto=route* in the */etc/ipsec.conf* file.

Task 15: Host-to-host transport mode VPN with PSK authentication

Configure Server A and Server B to establish a host-to-host transport VPN using PSK and IKEv2. You don't need Client A and Client B for this task.

To verify your setup start Wireshark in Server A and in Server B. On server A, ping the IP address of server B on the public network – 192.168.70.6.



In the terminal, you should see replies from 192.168.70.6. In Wireshark (both on Server A and Server B) you should see first a number of ISAKMP packets exchanged between Server A and Server B. This is the IKE protocol used to negotiate keys and cryptographic parameters. Then, you should see ESP traffic, with source IP and destination IP addresses alternating between 192.168.70.5 and 192.168.70.6.

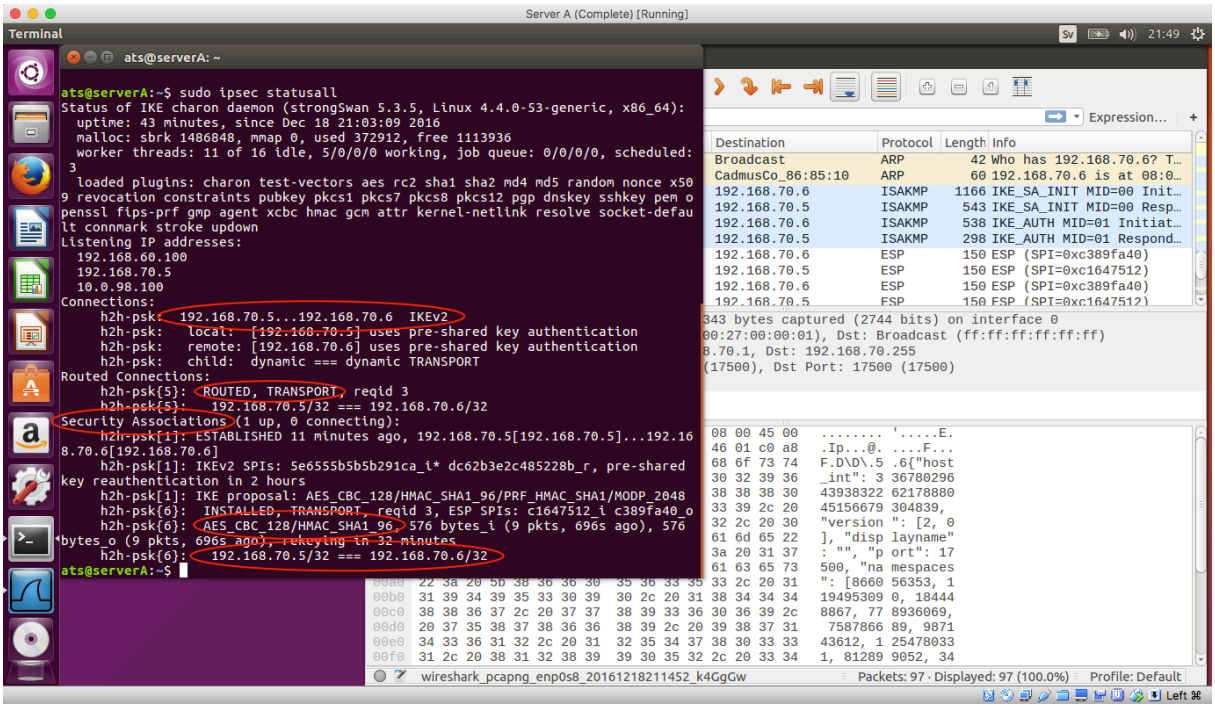
Include a similar screenshot and the configuration from *ipsec.conf* and *ipsec.secrets* in the report.

You can also ask strongSwan to report the established connections by entering the following command[14] on either server A or server B:

sudo ipsec statusall

You should an output similar to the one shown below.

[14] *man ipsec*



But how can you be sure that the ESP traffic observed in the previous task is really ICMP traffic? Luckily, Wireshark can actually decrypt recorded traffic if provided with the cryptographic parameters and the correct keys.

Keep in mind that strongSwan does dynamic keying, which means it changes the encryption keys frequently. You will want to capture some fresh ping traffic with Wireshark before moving to the next task.

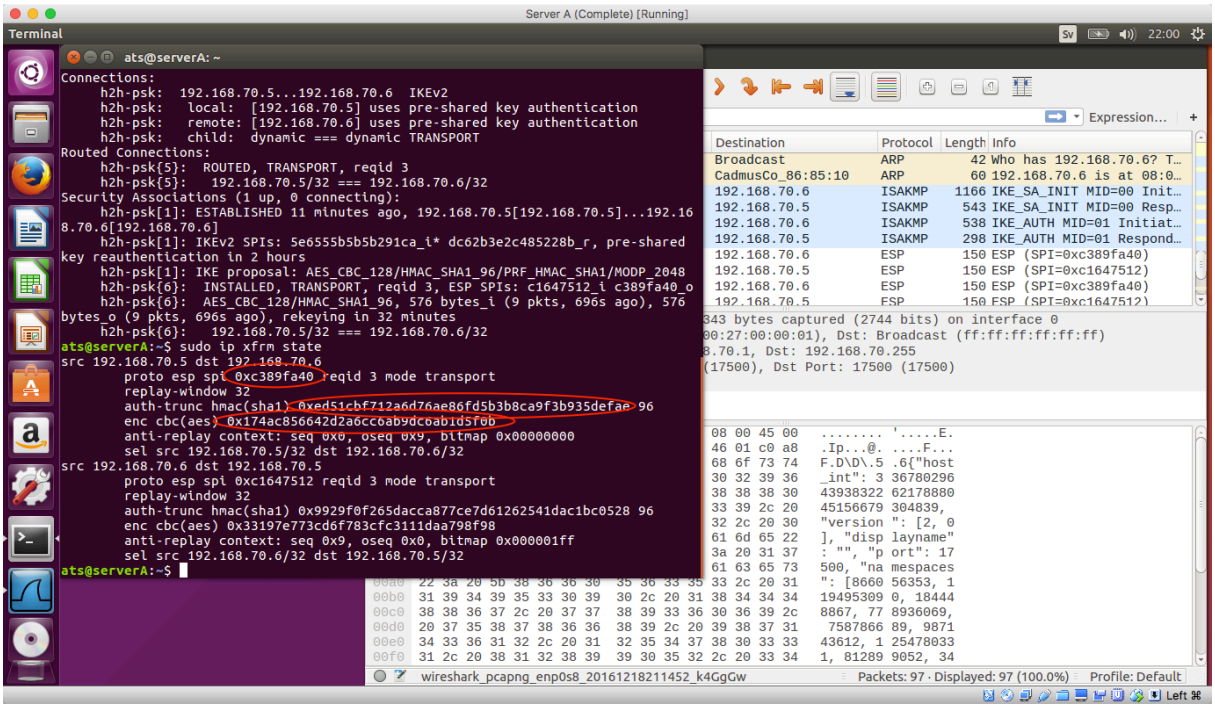
Task 16: Decrypt traffic with Wireshark

You need to extract the SPI numbers, the encryption keys and the HMAC secrets used in the established SA. You need to do this in the same VM where you will decode the traffic. Enter the following command[15]:

sudo ip xfrm state

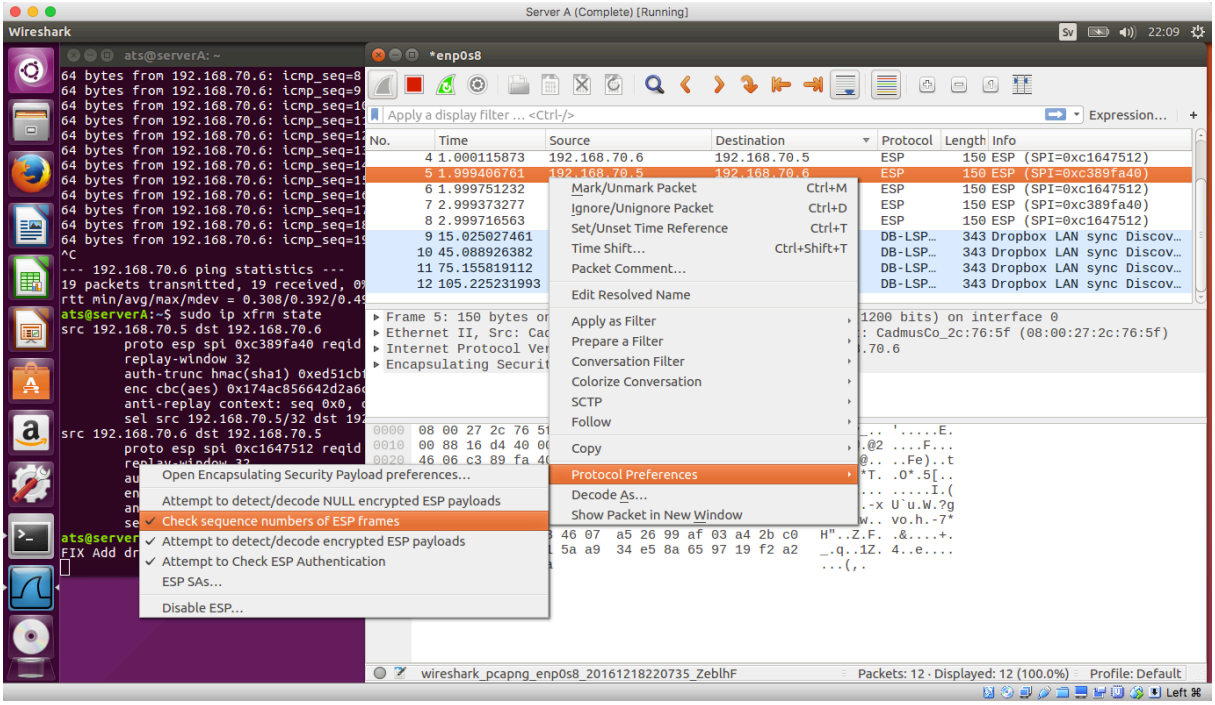
You should see output similar to the one shown below:

[15] *man ip-xfrm*

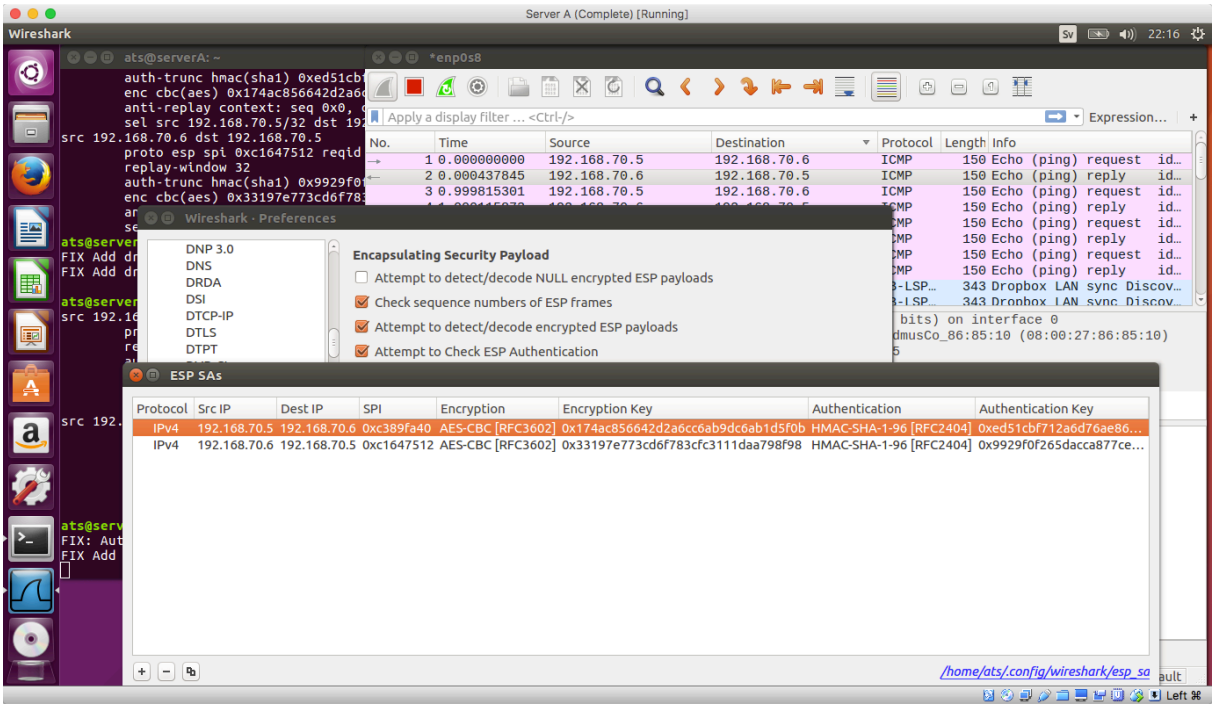


If you look carefully, you'll see that two SAs were established: one for the traffic 192.168.70.5 to 192.168.70.6 and one for the opposite direction. You need to add both to Wireshark if you want to see packets in both directions!

Right-click on an ESP packet in Wireshark and select Protocol Preferences -> ESP SAs…

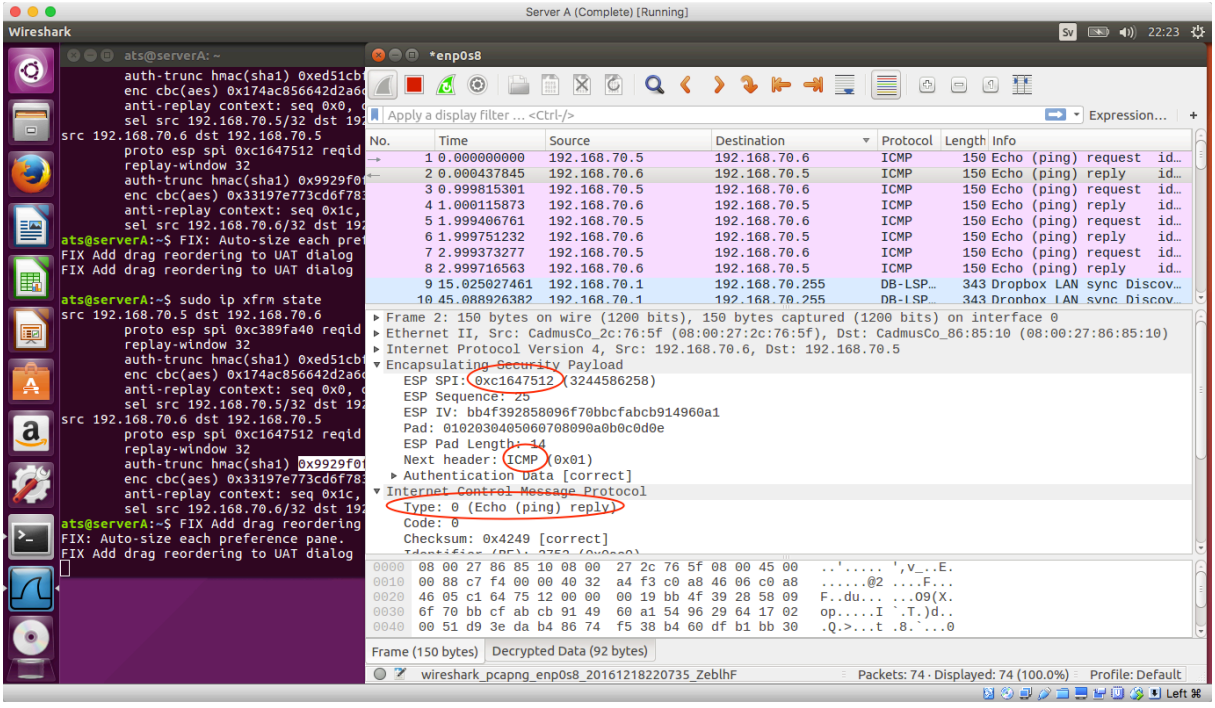


Add the two SAs to the new dialog window and cut and paste the necessary values from the terminal. It should look similar to the image below:



The type of encryption and authentication algorithm used is visible both in the output from the *ipsec statusal*l command as well as from the output of the *ip xfrm state* command. The algorithm name is clearer in the output of the first command.

You should be able to see in Wireshark decrypted ICMP packets. Select one and expand ESP and ICMP in the protocol view as shown below. The SPI number should match the one from the terminal.



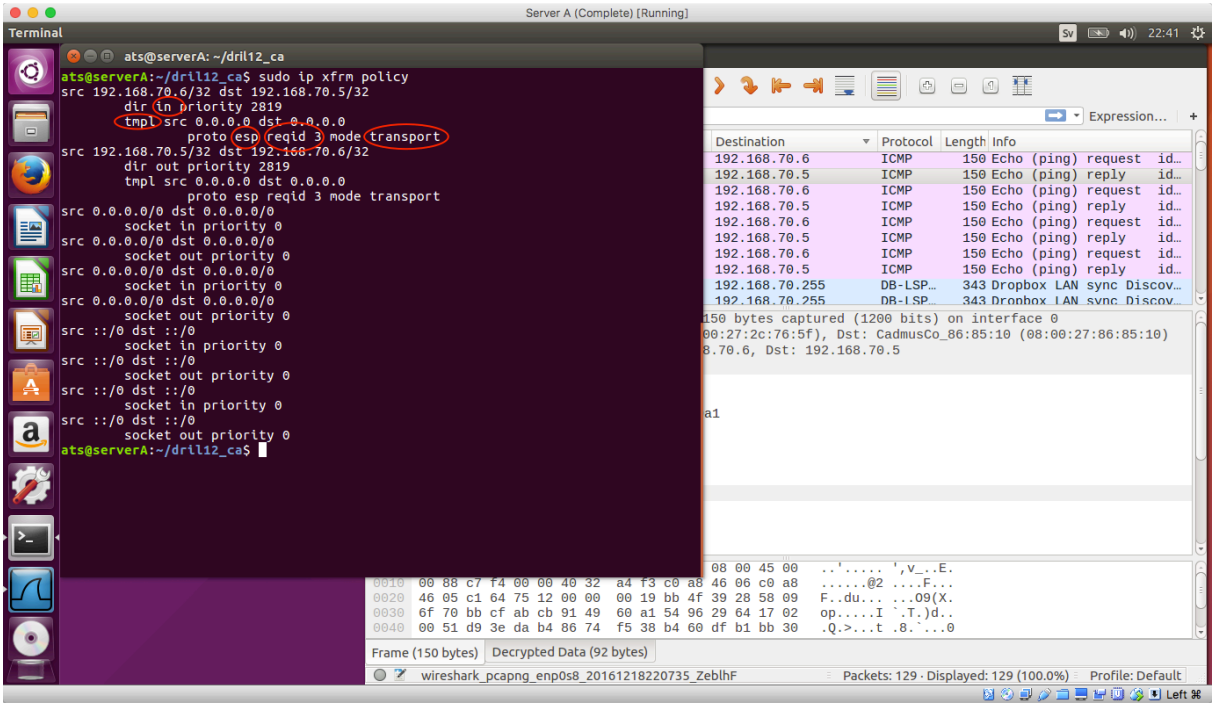
Include in your report the output from *ipsec statusall*, *ip xfrm state*, a screenshot where you show the configured ESP SAs in Wireshark and another one where you show the details of a decrypted ICMP packet (the ESP SPI and the Nest header in particular).

You will now work towards understanding the connection between the SPD and the SAD.

Task 17: List the entries in the SPD

Use the following command[16] to list the entries in the SPD:

sudo ip xfrm policy



You should see something similar to what is shown in the figure above. Some of the important fields have been encircled. The *reqid* field connects an entry in the SPD with the corresponding entry in the SAD. You can see that by comparing this output to the output of *the ip xfrm state* command shown above, where the same reqid value appears together with the same IP address pair. The *tmpl* keyword defines the beginning of a template that specify how IPsec should encapsulate matching packets.

Explain in your report what is the purpose of the other encircled fields (keywords) in the figure above. What is the specific purpose of the other entries in the SPD?

Host-to-host transport mode VPN with certificate authentication

You'll now improve the security of the VPN by changing the strongSwan configuration from PSK authentication to certificate authentication. You will make use of the knowledge from the OpenSSL part of this lab to create certificates for Server A and Server B using the *openssl* command (**not** the *ipsec pki* command!). Use the information from https://wiki.strongswan.org/projects/strongswan/wiki/SimpleCA to install the certificates in strongSwan.

[16] *man ip-xfrm*

The *ca1* certificate authority developed earlier on Server A will be used for generating certificates. The PSK VPN implemented in Task 15 should be exploited to copy the necessary cryptographic material between Server A and Server B (e.g., you can use the scp command to copy from 192.168.70.5 to 192.168.70.6 and vice-versa).

Task 18: Host-to-host transport mode VPN with cert authentication

Use *ca1* to sign server certificates for Server A and Server B as you have done earlier for Apache. When asked for Common name (CN), make sure that you enter the server's IP address from the public network: 192.168.70.5 for Server A and 192.168.70.6 for Server B. Use the IP addresses as the <name> part of the generated cryptographic material.

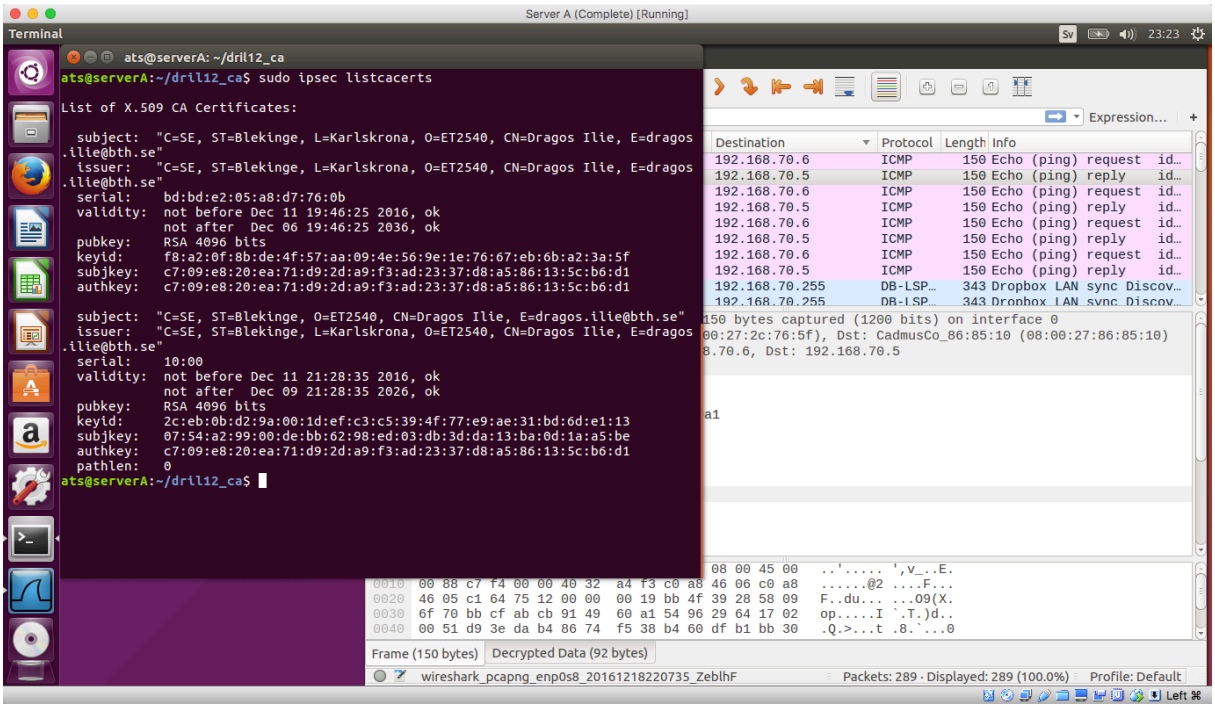
strongSwan does not know how to handle certificate chains contained in a single PEM file. Instead, you need to copy the individual certificates for the root CA and the intermediate CA to the strongSwan directory for CA certificates. Also, you need to either reboot or force strongSwan to read the CA certs. You do that using the command

sudo ipsec rereadcacerts

To list the discovered CA certs enter the command:

sudo ipsec listcacerts

You should see something like what is shown in the image below.



When an IKEv2 (Charon) negotiates as SA, it uses the subject distinguished name (DN) from the certificate to identify itself to the peer. If the peer does not recognize the ID, it drops the connection. You should be able to see the proposed ID for failed connections in the logs (if you have increased the log level as directed above).

You must set both *leftid* and *rightid* parameters in *ipsec.conf* to match the subject DN in the corresponding certificate.

Include in your report the modifications to the *ipsec.conf* and *ipsec.secrets* files. Add also evidence that you have configured the VPN as requested (transport mode and certificate authentication) and that traffic is encrypted. The evidence should be in the form of screenshots and output from the commands presented earlier.

Site-to-site tunnel mode VPN with certificate authentication

Now you will modify the strongSwan configuration to use tunnel mode. The goal is to automatically provide the VPN service to all nodes in Site A and Site B. However, we'll start small and the first attempt will be to establish a tunnel between Server A and Server B.

Task 19: Tunnel mode VPN with cert authentication between Server A and Server B

Configure strongSwan to setup an IPsec tunnel between Server A and Server B over the public network. Server A and Server B must be able to communicate over the tunnel using IP addresses from the private network. This can be tested by pinging the private addresses of Server A (192.168.60.100) and Server B (192.168.80.100), respectively.

You should be aware that in Wireshark you will see both plain-text and encrypted packets for inbound traffic but only encrypted (ESP) packets for outbound traffic (the plain-text outbound packets don't reach Wireshark prior to encryption). This is normal behavior and a consequence of how packet capture is implemented in the network stack.

Include in your report the modifications to the *ipsec.conf* and *ipsec.secrets* files. Add also evidence that you have configured the VPN as requested (tunnel mode and certificate authentication) and that traffic is encrypted. The evidence should be in the form of screenshots and output from the commands presented earlier.

Task 20: Tunnel mode VPN with IP forwarding for client A and client B

For this task, you need to start Client A and Client B VMs in addition for Server A and Server B. You must configure Server A and Server B to forward packets between Client A and Client B over the VPN tunnel established in Task 19. This is similar to the task from the Iptables lab.

Include in your report the modifications (if any) to the *ipsec.conf* and *ipsec.secrets* files. Add also evidence that you have configured the VPN as requested (tunnel mode and certificate authentication) and that traffic between Client A and Client B is transported over the IPsec tunnel established between Server A and Server B. The evidence should be in the form of screenshots and output from the commands presented earlier.

Task 21: Site A to Site B VPN with default DROP firewall rules

Change the *iptables* default policy to DROP on server A and server B. Modify your *iptables* rules from the previous lab to allow Client A and Client B to communicate with each other

over the tunnel. Client A and Client B must also be able to reach the Internet over the NAT interface of Server A and Server B, respectively.

Include in your report the modifications (if any) to the *iptable.sh* script. Add also evidence that you have configured the VPN as requested (tunnel mode and certificate authentication) and that traffic between Client A and Client B is transported over the IPsec tunnel established between Server A and Server B. Show also evidence that Client A and Client B can reach the Internet via Server A and Server B, respectively. The evidence should be in the form of screenshots and output from the commands presented earlier.

Deliverables

You are expected to provide an **individual** report in PDF format, where you detail how you solved each task included in the lab guide. Provide screenshots as proof that your commands/changes work as intended. In tasks that required you to make configuration changes or modify firewall rules, list the modifications you made and explain where you made them (on which VM and in what file).

Your report must be written in English. On the first page make sure you include your full name (as it appears in Canvas), your personal number, and your e-mail address.

In addition to the report, upload also your firewall script, the *openssl.cnf* files and the *ipsec.conf* and *ipsec.secrets* files for Server A and Server B, as they are after solving Task 27. Include also any additional files you may have changed to enabled requested functionality (like forwarding), such as */etc/network/interfaces* or */etc/resolv.conf*.