Lesson #9 - Cryptography in GNU/Linux and secured communication

Advanced Linux Administration

Karel Šrot

Motivation

- Be able to verify peer’s identity
- Prove the authorship, non-repudiation
- Securely transfer data over the network

Encryption and Decryption

Encryption - The process of encoding data in such a way that only authorized parties can understand the information.

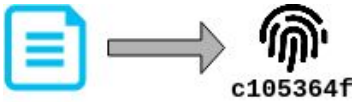
Decryption - The process of decoding encrypted information.

Cryptographic Key - A secret used to encrypt and/or decrypt information.

Cipher - An algorithm for performing encryption or decryption. Usually a key is required and the procedure of encryption/decryption is varied depending on the key.

Cryptographic hash function

A function designed to take a string of any length as input and produce a fixed-length hash value without revealing any information about the input.



Additionally, cryptographic hash function must have the following properties:

- It is computationally efficient
- It is deterministic - for a given input always provide same result
- It is infeasible to generate a message that yields a given hash value
- It is infeasible to find two different messages with the same hash value
- A small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value (avalanche effect)

Cryptographic hash function - Examples

MD5 - designed in 1991, produces hash digest of 128 bits, insecure

SHA-1 - published in 1993, produces a hash digest of 160 bits, insecure

SHA-2 - set of hash functions, basically based on SHA-256 and SHA-512 modified to provide different lengths of hash digest (224, 256, 384, 512)

Bcrypt - published in 1999, based on blowfish cipher, a password hashing function with arbitrary slow computation, used for password storage in some BSD/Linux distributions, most implementations produces hash digest of 184 bits

RIPEMD-160, WHIRLPOOL, SHA-3, ...

Cryptographic hash function - Usage

Cryptographic hash functions are being used for:

- Password verification (/etc/shadow)
- Data integrity checking
- Message Authentication Codes
- Digital signatures
- Pseudo-random number generators,...

Command-line utilities: `sha256sum, sha512sum, sha1sum, md5sum, ...`

Collision examples:

- MD5 collision - example1, example2
- SHA-1 collision - example3

Symmetric and Asymmetric cryptography

Symmetric key cryptography

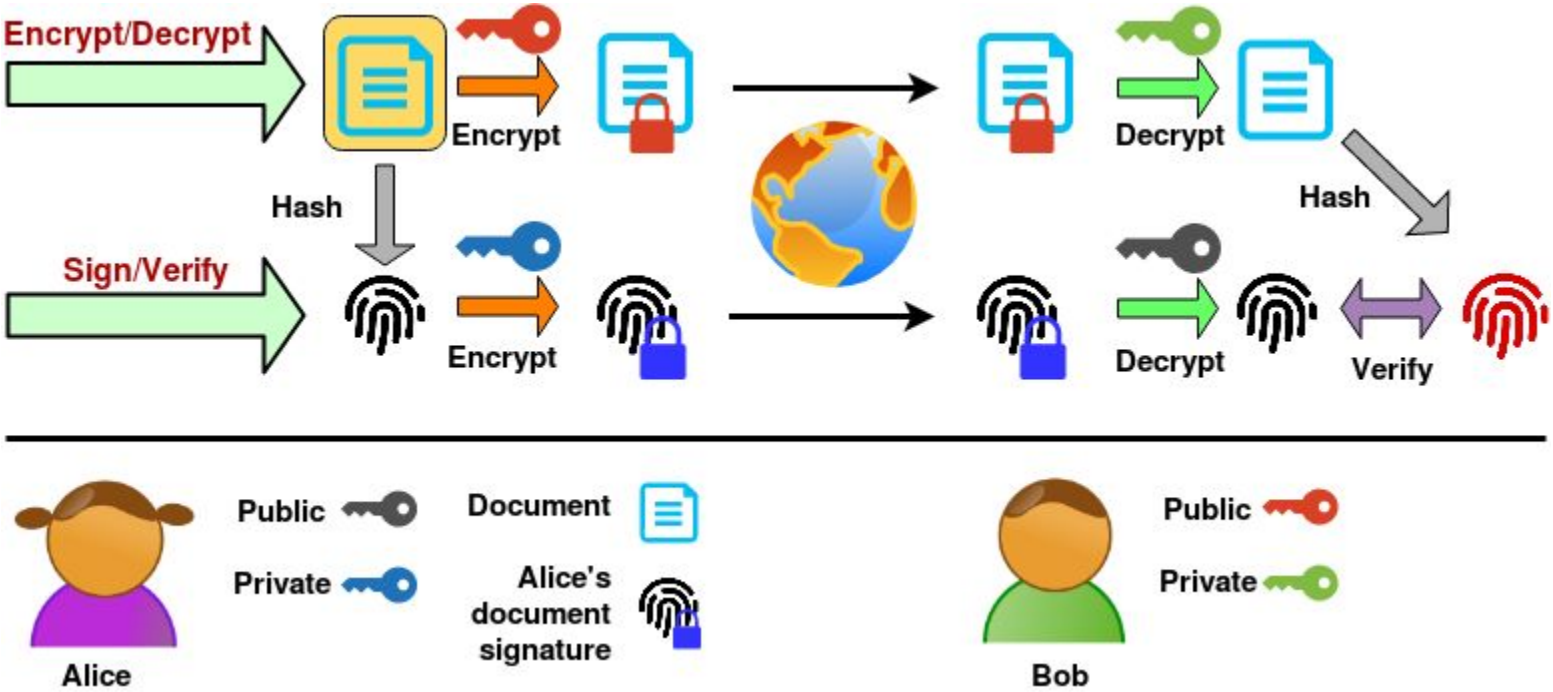
The same key is used both to encrypt and decrypt messages. It is mostly used in data encryption.

Asymmetric cryptography, Public key cryptography

Uses pair of keys - **Public key** which may be disclosed and **Private key** which is known only to the owner. It is utilized to provide confidentiality, non-repudiation, it is also used in authentication, digital signature, (symmetric) key exchange, ...

Symmetric cryptography is way faster but its weakness is the key distribution. In practice a combination of both approaches is used.

Public key cryptography - Data encryption and signing



Symmetric ciphers - Stream ciphers

Encrypts one bit/byte of data at a time using the generated pseudo-random cipher data stream (keystream).

- **One-Time Pad** - Invented in 1917 by Gilbert Vernam, perfectly secure, however the key must be at least as long as the message
- **RC4** - Designed in 1987 by Ron Rivest but kept secret until 1994, one of the most popular ciphers (used in TLS, WEP/WPA,...)
- **Salsa20** - Designed in 2005 by Daniel Bernstein
- **Chacha20** - Designed in 2008, based on Salsa20, widely used and often replacing RC4 (TLS, OpenSSH, *BSD/Linux random number generator, VPN,...)

Symmetric ciphers - Block ciphers

Works on fixed-length blocks of data, encrypting blocks one by one. Often combine data from different blocks in order to provide additional security.

- **Triple DES (3DES)** - Invented in the early 1970s, based on DES cipher which became insecure, still widely used, uses 64-bit blocks, keys 56/112/168 bits
- **Advanced Encryption Standard (AES)** - Developed in 1997 by Vincent Rijmen and Joan Daemen, one of the most popular ciphers in the world, uses 128-bit blocks, key 128/192/256 bits
- **Blowfish** - Designed in 1993 by Bruce Schneier, also popular, uses 64-bit blocks, keys from 32 up to 448 bits
- **RC2, Camellia, Serpent,...**

Asymmetric algorithms

- **RSA** - designed by Ron Rivest, Adi Shamir and Leonard Adleman in 1977, one of the most popular algorithms with public key encryption. It can be used for either encryption of messages or for digital signatures, secure keys have length 2048, 3072,...
- **Diffie-Hellman key exchange** - published by Whitfield Diffie and Martin Hellman in 1976, a method of securely exchanging cryptographic keys over a public channel
- **Elliptic-curve cryptography (ECC)** - requires smaller keys compared to non-EC cryptography, 256-bit public key should provide comparable security to a 3072-bit RSA public key
- **ElGamal, DSS/DSA, ...**

Public key certificate

Public key certificate / digital certificate / identity certificate is an electronic document used to prove the ownership of a public key.

Includes information about the key, information about the identity of its owner, and the digital signature of an entity that has verified the certificate's contents.

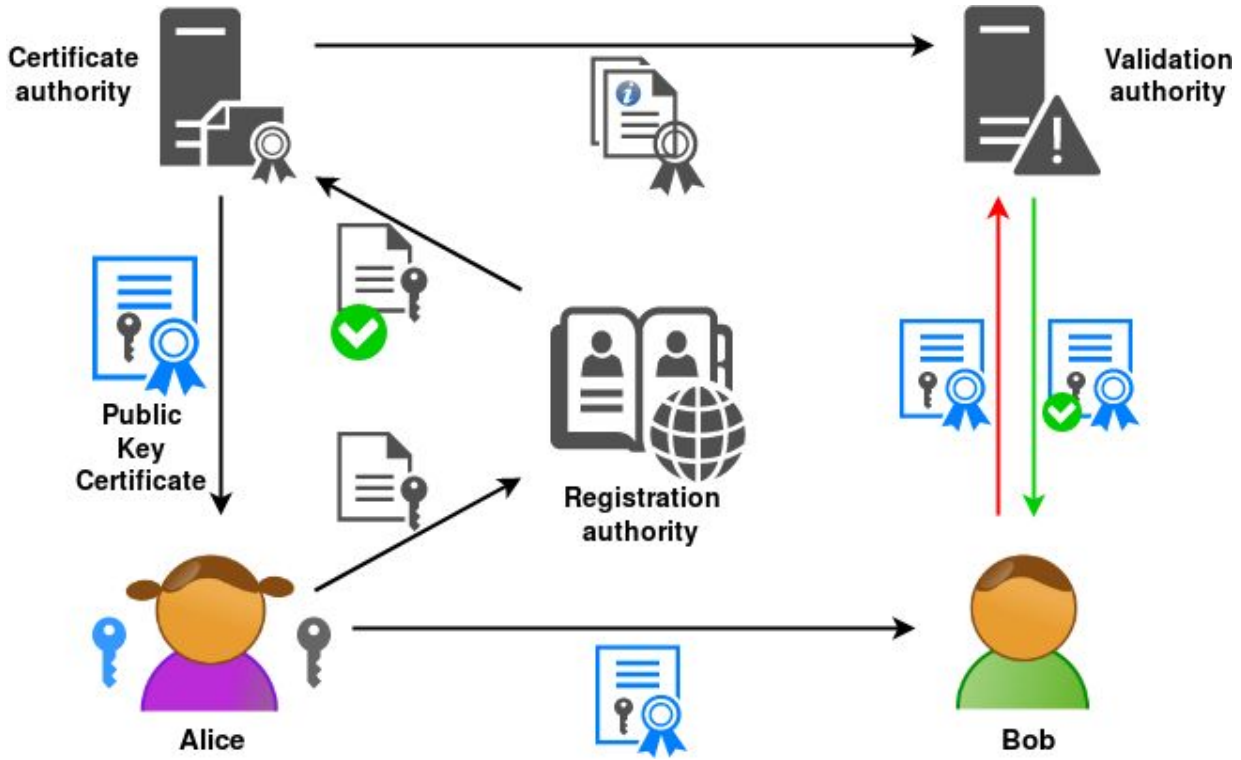
If the signature is valid, and the software examining the certificate trusts the issuer, then it can use that key to communicate securely with the certificate's subject.

Public key infrastructure

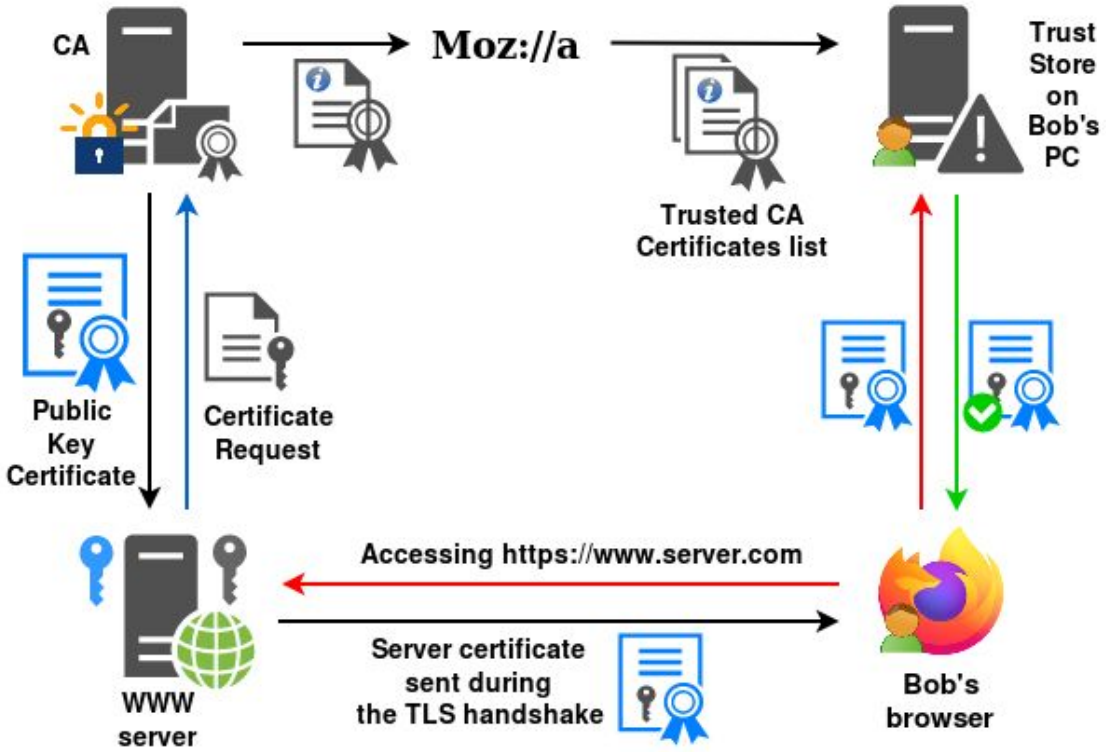
PKI is a set of roles, policies, HW, SW and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. **PKI** consists of:

- **Certificate authority (CA)** - stores, issues and signs the digital certificates
- **Registration authority (RA)** - verifies the identity of entities requesting digital certificates
- **Validation authority (VA) -** can provide this entity information on behalf of the CA.
- **Certificate management system** - managing things like the access to stored certificates or the delivery of the certificates to be issued
- **Certificate policy** stating the PKI's requirements concerning its procedures to allow outsiders to analyze the PKI's trustworthiness.
    - **Certificate Revocation List (CRL)** - contains serial numbers of revoked certificates.
    - **Online Certificate Status Protocol (OCSP)** - used for certificate revocations, OCSP servers can be consulted by clients to check if the server certificate has been revoked.

PKI - Certificate flow theory



PKI - Certificate flow real-life example



X.509 certificate

The most common format for public key certificates. Because X.509 is very general, the format is further constrained by profiles defined for certain use cases.

The structure of an X.509 v3 digital certificate:

Certificate
- Version Number
- Serial Number
- Signature Algorithm ID
- Issuer Name
- Validity period
  - Not Before
  - Not After
- Subject name

Certificate - cont.
- Subject Public Key Info
  - Public Key Algorithm
  - Subject Public Key

Certificate (optional)
- Issuer Unique Identifier
- Subject Unique Identifier
- Extensions

...

Certificate Signature Algorithm
Certificate Signature

```
$ man x509v3_config
```

X.509 certificate - Example

Certificate:
Data:
Version: 3 (0x2)
Serial Number:
dd:d9:04:b0:a1:05:c2:75:02:00:00:00:00:47:d8:7a
Signature Algorithm: sha256WithRSAEncryption
Issuer: C=US, O=Google Trust Services, CN=GTS CA 1O1
Validity
Not Before: Oct 10 20:54:46 2019 GMT
Not After : Jan 2 20:54:46 2020 GMT
Subject: C=US, ST=California, L=Mountain View, O=Google LLC, CN=*.google.com
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
Public-Key: (2048 bit)
Modulus:
00:a6:f3:90:4e:25:d1:c3:33:77:34:22:ea:63:d4:
e2:dc:a8:67:41:7f:91:8c:a9:0e:08:26:56:c3:62:
...
Exponent: 65537 (0x10001)
X509v3 extensions:
X509v3 Key Usage: critical
Digital Signature, Key Encipherment

X509v3 Extended Key Usage:
TLS Web Server Authentication
X509v3 Basic Constraints: critical
CA:FALSE
X509v3 Subject Key Identifier: 4F:9E:BA:11:59:F3:CE:8D...
X509v3 Authority Key Identifier: keyid:98:D1:F8:6E:10:EB:CF:...
Authority Information Access:
OCSP - URI:http://ocsp.pki.goog/gts1o1
CA Issuers - URI:http://pki.goog/gsr2/GTS1O1.crt
X509v3 Subject Alternative Name:
DNS:*.google.com, DNS:*.android.com, ...
X509v3 Certificate Policies:
Policy: 2.23.140.1.2.2
Policy: 1.3.6.1.4.1.11129.2.5.3
X509v3 CRL Distribution Points:
Full Name:
URI:http://crl.pki.goog/GTS1O1.crl

Signature Algorithm: sha256WithRSAEncryption
a0:55:39:1c:5a:d0:f1:45:69:4a:18:ce:89:7e:56:0e:a4:c4:
94:26:03:73:3d:7a:2d:8e:6e:36:e2:91:b6:d2:8c:26:52:20:
...

X.509 - generating custom certificates

Generating Self-signed (CA) certificate along with a key at one step:

```
$ openssl req -x509 -newkey rsa:4096 -nodes -keyout CA.key -out CA.crt -days 365
```

Generating a Key:

```
$ openssl genrsa -out mydomain.com.key 2048
```

Generating a certificate signing request:

```
$ openssl req -new -sha256 -key mydomain.com.key -subj "/C=CZ/O=MyCompany/CN=mydomain.com" -out mydomain.com.csr
```

Generating a CA signed certificate based on the CSR:

```
$ openssl x509 -req -in mydomain.com.csr -CA CA.crt -CAkey CA.key -out mydomain.com.crt -days 500 -sha256 -CAcreateserial
```

More detailed HowTo.

Managing Trusted certificates in a Linux system

Unfortunately, various crypto libraries are having custom (default) location and format for storing trusted certificates, .e.g.

- OpenSSL - `/etc/pki/tls/cert.pem, /etc/pki/tls/certs/`
- NSS - `/etc/pki/nssdb/, /usr/lib64/libnssckbi.so in NSSDB format`
- JAVA - `/usr/lib/jvm/java-*-openjdk-*/jre/lib/security/cacerts` in JKS keystore

Additionally, some applications have to be also configured explicitly, e.g. Firefox.

In some distributions **Shared System Certificates** facility can be used overcome this inconsistency.

Shared System Certificates

System-wide trust store enabling NSS, GnuTLS, OpenSSL, and Java to share a default source for retrieving system certificate anchors and black list information. By default, contains the Mozilla CA list, including positive and negative trust.

In Fedora, the consolidated system-wide trust store is located in

- `/etc/pki/ca-trust/`
- `/usr/share/pki/ca-trust-source/` - processed with lower priority

Certificate files are treated depending on the subdirectory they are installed to:

- `anchors` - for trust anchors, i.e. root/CA certificates
- `blacklist` - for distrusted certificates

To add/blacklist a certificate, copy a respective certificate file to a respective subdirectory and run the `update-ca-trust` command (more detailed guide).

Managing enabled Cipher suites in a Linux system

Similarly to Trusted certificates, crypto libraries differ in a way how enabled cipher suites are configured.

Also, crypto algorithms are becoming insecure as time goes by.

Problem:. How to keep consistent cipher suite configuration across all the crypto libraries used on the system?

In some distributions a system-wide **Crypto Policy** can be configured supporting the most common crypto libraries and even some applications.

System-wide Crypto policy

Allow setting a consistent security profile for crypto on all applications in a system.

A profile specifies details like acceptable TLS/SSL versions, the acceptable ciphersuites and the preferred order, acceptable parameters in certificates and key exchange, etc.

There are predefined profiles available, such as LEGACY, DEFAULT and FUTURE.

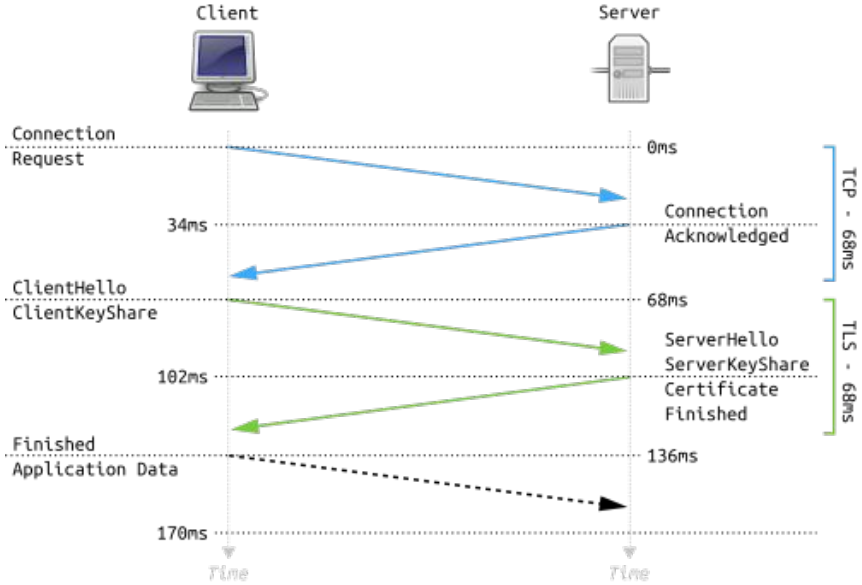
An administrator configures the system profile using the `update-crypto-policies --set PROFILE` command from a package `crypto-policies-scripts`

Resources: man crypto-policies, Fedora CryptoPolicy wiki, project README

TLS - Transport Layer Security

Security protocol designed to facilitate secure communication over the Internet - providing authentication, privacy and data integrity.

Used for encrypting the communication between web applications and servers, but can also be used to encrypt other communications.



During the course of a TLS handshake, the client and server together will agree on TLS protocol version, cipher suites to use, do authenticate each other using certificates and generate session key for symmetric encryption.

SSH key-based authentication

Reliable and more secure alternative to password-based authentication.

Additionally allows automated, passwordless login.

Private key is stored in `~/.ssh/id_rsa` (or `id_dsa`, `id_ecdsa` etc.) on your local system. Your public key is typically stored in `~/.ssh/id_rsa.pub`.

To configure key-based authentication on a remote system add your public key to the `/home/USERNAME/.ssh/authorized_keys` file on the remote system.

SSH-agent tool can be used to manage repetitive key unlocking in case the key is password protected.

SSH key-based authentication - Example

Generate a new key pair with ssh-keygen command

- `$ ssh-keygen -t rsa -b 4096`

Upload your public key to a remote system, enabling key-based authentication

- `$ ssh-copy-id -i ~/.ssh/id_rsa user@host`

Run the `id` command on a remote system `host`

- `$ ssh user@host ‘id’`

Workshop

60 min

Lab 1 - Using hash functions for checksums [5 min]

1. For some of the collisions illustrated at websites referred below compute beside MD5/SHA-1 also SHA256 and SHA512 checksums.

- MD5 collisions - example1, example2
- SHA-1 collision - example3

Hint: To get MD5 and other checksums of a file available at given URL you can use command similar to:

- `$ curl -s 'SOME_URL' | md5sum`

Lab 2 - Configure httpd with custom TLS certificate [15 min]

1. Generate a key (CA.key) and certificate (CA.crt) for your custom Certificate authority, use 4096-bit long key.
2. Generate a signed certificate for your domain `testdomain.com` (refer the domain name in the **CN** field!), use 2048-bit long key.
3. Review both certificates using the `openssl x509 -in CERTIFICATE.crt -text` command. When are these certificates going to expire?
4. As we do not really own `testdomain.com` domain, add following line to `/etc/hosts`
   - `127.0.0.1  testdomain.com`

   Verify with `$ ping testdomain.com`
5. Install `httpd` and `mod_ssl`. Configure TLS in httpd
   - `# cp testdomain.com.key /etc/pki/tls/private`
   - `# cp testdomain.com.crt /etc/pki/tls/certs`
   - Edit `/etc/httpd/conf/httpd.conf` and update `ServerName` option with your domain
   - Edit `/etc/httpd/conf.d/ssl.conf` and update `SSLCertificateFile` and `SSLCertificateKeyFile` options with proper file paths

Lab 2 - Configure httpd with custom TLS certificate - cont.

6. Prepare a web page content
   - `# echo hello > /var/www/html/index.html`
7. Start httpd service
8. Verify that HTTP works and ‘hello’ is printed
   - `$ curl http://testdomain.com`
9. As our CA.crt is not trusted, following command should fail complaining on the unknown certificate issuer
   - `$ curl https://testdomain.com`
10. Pass CA.crt as trusted to curl directly and verify that ‘hello’ is printed
    - `$ curl --cacert CA.crt https://testdomain.com`

Lab 3 - Configure new CA certificate as trusted [10 min]

1. Configure the previously generated CA certificate as a trusted anchor. Update the system trust store configuration.
2. List trusted certificates using `trust list` command and confirm your CA is present
3. Verify that the CA certificate is now trusted and curl `--cacert` option is not needed any more
   - `$ curl https://testdomain.com`
4. Remove your CA certificate from trusted anchors and update system trust store configuration.
5. Verify that following command is failing again
   - `$ curl https://testdomain.com`

Lab 4 - Change the system-wide Crypto policy [10 min]

1. Confirm that the following command works
   - `$ curl --cacert CA.crt https://testdomain.com`
2. Change the system crypto policy to FUTURE.
3. Run the command above again and observe the result/error.
4. In `crypto-policies` man page search for the minimal acceptable RSA key-size for the FUTURE profile
5. Regenerate the server certificate (not CA) with a larger RSA key-size and sign it. Copy both the certificate and the key to the appropriate location and restart httpd service. Confirm that curl command works now.
6. Set the system crypto profile back to DEFAULT.

Lab 5 - Change user password encryption method [5 min]

1. Add a user accounts `alice` to your system and set the user password.
   - `# useradd -m USER; passwd USER`
2. Edit `/etc/pam.d/system-auth` and on a line with `pam_unix.so` configuration replace `sha512` with `blowfish`. Be sure not to make any other changes!
3. Add a new user account `bob` to your system and set the user password.
4. As `root`, in `/etc/shadow` observe the content of lines storing encrypted password for `alice` and `bob`.

Lab 6 - Configure key-based authorization for SSH [5 min]

In this lab we would use `localhost` as a substitute to a remote system.

1. As a user `alice` generate a new `ecdsa` key pair for SSH
2. Configure `alice`'s SSH key for a password-less login to `bob`'s account on the `localhost` system
3. Verify that the key-based authentication works by running
   - `$ ssh bob@localhost 'id'`
4. As a user `bob` on `localhost` review the content of `~/.ssh/authorized_keys` file.
5. Is the password required when `alice` tries to login to `alice@localhost`? Why?
6. Disable password authentication for the user `bob` by running
   - `# passwd -l bob`

   Review the respective line in `/etc/shadow`. What is the difference?
7. Confirm that the SSH key-based authentication still works.