1) DIGITAL SIGNATURE

You are asked to implement two functionalities in the system: (1) to sign any data handled by the system and (2) to verify such signature. The signature must persist, so you can store it in a file (by itself, the most immediate option) or, if it suits you better, you can store it in a DB or JSON (with great care not to alter anything or at least to ensure that possible encodings or representations in other formats are completely reversible without leaving a trace).

To be able to sign, you must first create a pair of private/public keys. I recommend using RSA or EdDSA on the Curve25519.

1.a) RSA

- RSA key generation: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#generation
- RSA signing: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#signing

parameters for the padding of the signature: MGF1 with SHA256 and PSS with MAX_LENGTH (as in the first example)

- RSA (signature) verification: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#verification

Obviously you must use the same padding as when signing.

The message to sign must be in bytes and the signature returned is also in bytes.

To store the signature and then verify it you should store on one side the message and the signature (however you prefer, but so that you can recover exactly the bytes that were signed). The most immediate approach is to save that message and the signature in a file each one separately and write to the files in mode "wb" (write binary). If you store them in a database or json, you will need to encode them beforehand.

The public and private keys should also be saved in a file each (or in a database/json, with “great care”) so that they persist. To do this, since they are OBJECTS you should serialize them and then write them to the file in "wb" mode as well.

- Serialization of RSA private key and RSA public key: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#key-serialization

I suggest to use the following parameters to serialize the private key:

- PEM encoding
- PKCS8 format (I think this is the one that will be interoperable with PKI, but I’m not 100% sure; if not, TraditionalOpenSSL)
- *BestAvailableEncryption* and you pass the password through a variable as a parameter. If it is the system's password you can write it above in the code (main) with many '****' or, preferably, you request the password from the user (as input) or you pass it to the system when you run the code from the command line, or retrieve it from the computer enviroment (eg, like PATH or similar).

For the public key:

- PEM encoding
- SubjectPublicKeyInfo format

Once you have them saved in the files, when you need to use the private or public key, you should read them from the file. Therefore, you should read (in "rb" mode) and deserialize the keys to recover the corresponding objects:

- Deserialization of RSA private key (referred to as "Key loading" on the library’s website):

https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#key-loading

https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#cryptography.hazmat.primitives.serialization.load_pem_private_key

- Deserialization of RSA public key: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#cryptography.hazmat.primitives.serialization.load_pem_public_key

1.b) Ed25519

All you need is in this section: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/ed25519/#ed25519-signing

---

2) PUBLIC KEY INFRASTRUCTURE (PKI)

Just as you should keep private keys encrypted, the public keys must be certified, either by yourself (self-signed certificate) or by a Certification Authority (part of a PKI).

Certificates are of the X.509 type (consult https://en.wikipedia.org/wiki/X.509). You should also save them to a file (using the "wb" mode) after serializing them (to serialize the certificate object you only need to run the public_bytes() method of the certificate itself and it returns the byte string you have to save: https://cryptography.io/en/latest/x509/reference/#cryptography.x509.Certificate.public_bytes).

And when you want to use the public key you should start from the certificate file, read the contents (using "rb") and deserialise the certificate object using the appropriate PEM function https://cryptography.io/en/latest/x509/reference/#cryptography.x509.load_pem_x509_certificate

2.a) SELF-SIGNED CERTIFICATE OPTION (WITHOUT PKI)

You should follow the library's tutorial: https://cryptography.io/en/latest/x509/tutorial/#creating-a-self-signed-certificate

In the example (after the key pair is created) first fill in the NAME (which is the same for the certificate subject—the holder—and the issuer of the certificate—because it is self-signed). Then instantiate the CertificateBuilder class to pass the certificate data and finally generate the signature (passing the private key used to sign at the end—the "key" parameter). You may omit the extensions.

2.b) CERTIFICATE GENERATED BY A CA

For the PKI (CA) we will use the command-line tools offered by the OpenSSL library, specifically those for creating a demo PKI: ca, x509, req and verify.

- OpenSSL: https://www.openssl.org and https://www.openssl.org/docs/manpages.html

Windows binary downloads for OpenSSL (generally if you have Linux or Mac it will already be installed): https://wiki.openssl.org/index.php/Binaries

Or alternatively you can download and unzip a portable binary (no install needed but it is old, hence with vulnerabilities) from the LibreSSL distribution: https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/

The latest Windows version listed there is 2.5.5 from 2017 (https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/libressl-2.5.5-windows.zip).

To deploy a mini-PKI you can follow the instructions provided in Practice 3 of the OCW cryptography course (earlier version):

- Course: https://ocw.uc3m.es/course/view.php?id=270 ; Labs: https://ocw.uc3m.es/mod/page/view.php?id=3344

You also need to download an openssl_CA1.cnf configuration file (text), at least.

You should follow the instructions provided in the statement to create an AC (AC1). You do not need to create AC2.

However, the part where the end entity (A in the statement, in your system, one of the signing entities) creates the CSR (Certificate Signing Request) you must do in Python — not necessarily inside your system; it can be separate code.

To create the CSR you can follow the library's tutorial: https://cryptography.io/en/latest/x509/tutorial/#creating-a-certificate-signing-request-csr

Once you have it created (very similar to creating the self-signed certificate, beware), serialize it as shown at the end of the example (csr.public_bytes()) and copy it to the corresponding folder of the OpenSSL-deployed demo PKI.

From here you can continue with the OpenSSL practice instructions, having the CA sign the requested certificate. The generated cert.pem file (which will be 01.pem or 02.pem... depending on the serial number) you copy back to your Python environment (i.e., where your code expects to find it; P) together with the CA certificate (ac1.pem)

2.c) FOR BOTH OPTIONS a) and b): CERTIFICATE VERIFICATION

When verifying the signature you must also verify the certificate of the associated public key and, if applicable, the certificate chain (the CA certificate).

To verify the certificate you must verify its signature. For this you need to extract from the certificate the following fields (consult the X.509 object reference: https://cryptography.io/en/latest/x509/reference/#x-509-certificate-object):

- cert_to_verify.signature
- cert_to_verify.tbs_certificate_bytes
- cert_to_verify.signature_hash_algorithm

and you must also check which padding was used to generate the signature (most likely PKCS1v15 > https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#cryptography.hazmat.primitives.asymmetric.padding.PKCS1v15, you have to pass it directly as asym_padding.PKCS1v15() provided you have imported cryptography.hazmat.primitives.asymmetric.padding as asym_padding)

And once you have all this, you can use the public key with which the certificate's signature was generated to verify it as a normal signature:

- https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#verification

With this, you should verify both the A certificate and the CA certificate if you generated it.

Last modified: Wednesday, 12 November 2025, 12:56 PM

?