Security of Computer Systems

Project Report

Author:
Adam, Zarzycki, 193243

Version: 2.0

Versions

Version	Date	Description of changes
1.0	11.04.2025	Creation of the document
2.0	08.06.2025	Added the “Solution” part, updated the “AuxiliaryApp” part

Table of contents

1. Information about the project

1.1. Project summary

The main goal of the project is to realize a software tool for emulating the qualified electronic signature, i.e. signing *.pdf documents. The goal is to fully emulate the process, including the hardware toll needed for person identification.[1]

1.2. Project requirements

PROJECT SUBMISSION – Presentation during classes		
	Task	Points
1	Generation of RSA keys, storing private key in a secure form – 2nd auxiliary application	3
2	Usage of hardware tool (pendrive with encrypted private key) during signing procedure, automatic key detection must be implemented	3
3	Generation of correct signature file – the modified *.pdf with signature details, associated with signed document	4
4	Presenting of correct and incorrect validation of signature by user B (pointing out resistance to document modification).	5
5	Presentation the functioning main and auxiliary applications during project submission	5
REPORTS – The report is evaluated only after project presentation		
6	Partial report (presentation only) for the control meeting (+ code, + presentation during classes)	5
	Minimal requirements: - Presentation: e.g. possibility of generating RSA keys (auxiliary application with GUI), basic project of main application (2 points). - Code in *GitHub* repository (3 points).	
7	Project report (+ code, pointing bibliography in the report)	15
	- Description of realised task (4 points). - Description of key application functionality, pointing out code fragments (4 points). - Code documentation using Doxygen (5 points). - pointing out the bibliography (1 points). - code in *GitHub* repository (no *zip archive allowed) (1 points).	

Fig 1. Project requirements

[1] the instruction and requirements provided on the enauczanie platform.

2. Summary of the tasks realized

My work on the project resulted in the creation of two applications - the auxiliary application (used for RSA key generation, from now on called “AuxiliaryApp” and the main one (handling PDF documents signing and their later validation, from now on called “Solution”).

As required in this project’s instruction, the AuxiliaryApp lets User A generate a pair of RSA keys 4096 bits in length, and saves the private key to the attached pendrive. The Solution then automatically detects this pendrive, loads the private key and starts the signing process, resulting in a PAdES-signed PDF file. Later on, User B can use User A’s public key to validate this signature.

Both of the applications’ documentations were generated using Doxygen and added to the Github repository, link to which was attached at the end of this repport.

3. AuxiliaryApp

3.1. Short description of the AuxiliaryApp

I prepared an auxiliary app (according to the requirements given in the project instruction). Its implemented functionalities are as follows:

- generation of a pair of 4096b RSA keys,
- obtaining a numerical PIN from the user,
- hashing of the private key with the AES algorithm, using 256b long SHA256 hash of said PIN,
- writing the encrypted private key to an attached pendrive,
- writing the public key to the project root directory,
- generation of a X.509 certificate (necessary for public key validation during signing),
- detection of a pendrive's presence/lack thereof,
- a GUI showing the user stages of execution taking place.

3.2. Simplified block diagram of the AuxiliaryApp

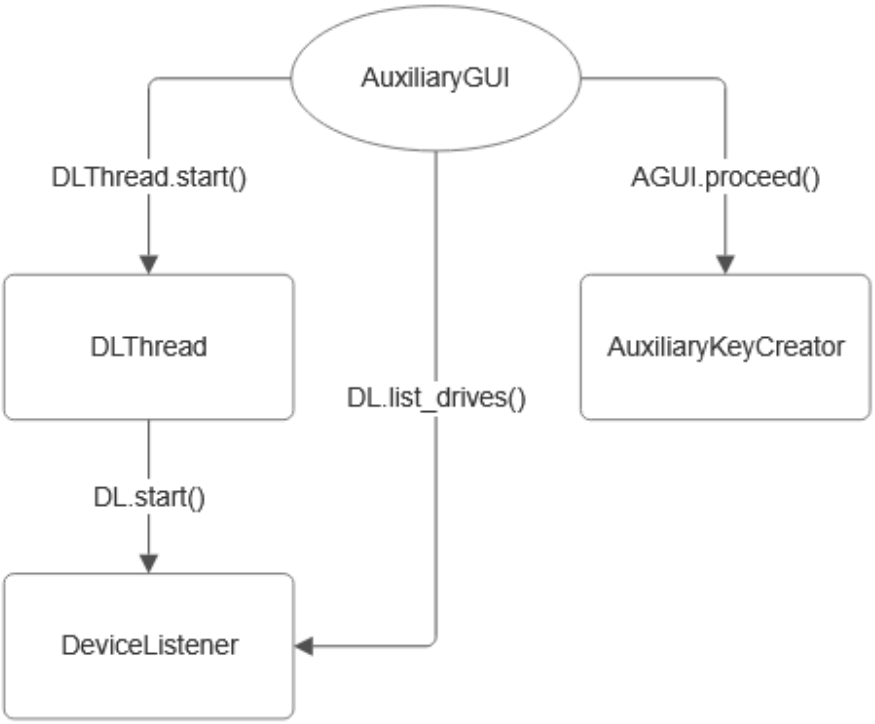

Fig 2. Simplified block diagram of the AuxiliaryApp

Explanation:

The singular point of entry to the AuxiliaryApp is the AuxiliaryGUI, which handles all the operations related to communicating with the user. When the user chooses to start the generation and encryption process, this class starts its proceed() method, which in turn invokes all the necessary methods of the AuxiliaryKeyCreator class, while also relaying the operation's progress to the user.

To determine whether the destination of encrypted private key - the pendrive - is currently available, the AuxiliaryGUI starts an additional DLThread (a descendant of the Thread class specially suited for device detection) with an instance of DeviceListener class inside. In case of an external device being attached/detached from the computer, it checks whether it is the desired pendrive, and changes the GUI status if so.

The AuxiliaryGUI instance also indirectly calls the list_devices() method of DeviceListener during setup phase, to determine the initial state of the pendrive.

3.3. Description of the most important methods

3.3.1. AuxiliaryGUI::on_devices_changed()

```
1 usage   ReadySetGet
def on_devices_changed(self):
    self._d_drive_comm.setText("Sprawdzam zmiany urzadzen zewnetrznych...")
    self._d_drive_comm.repaint()
    self._button.setEnabled(False)
    self._button.repaint()
    is_found = self.find_d_drive()
    if not self._is_d_drive_connected and is_found:
        self._is_d_drive_connected = True
        self._d_drive_comm.setText("Pendrive jest podpiety")
        self._button.setEnabled(True)
    elif self._is_d_drive_connected and not is_found:
        self._is_d_drive_connected = False
        self._d_drive_comm.setText("Pendrive nie jest podpiety")
        self._button.setEnabled(False)

    self._d_drive_comm.repaint()
    self._button.repaint()
    self.repaint()
```

Fig 3. AuxiliaryGUI::on_devices_changed()

This is the callback method of DeviceListener, it checks for an appearance/disappearance of the desired pendrive ("find_d_drive()" method), and changes the enabling of the starting button ("self.*_button") and the shown message ("self.*_d_drive_comm") accordingly.

3.3.2. AuxiliaryGUI::proceed()

```
self.show_current_arrow(self._current_stage_nr)
key_dict = {
    'key_priv': None,
    'key_pub': None
}
a = Thread(target=self._key_creator.generate_rsa_keys, args=(key_dict, ))
a.start()
while a.is_alive():
    QtWidgets.QApplication.instance().processEvents()
a.join()
key_priv = key_dict['key_priv']
key_pub = key_dict['key_pub']
self.set_texts()
time.sleep(0.5)

self.show_current_arrow(self._current_stage_nr)
pin_hash = self._key_creator.hash_pin_with_sha256(pin)
self.set_texts()
time.sleep(0.5)
```

Fig 4. Fragment of AuxiliaryGUI::proceed()

This method is the event function of the starting button, and it invokes all the required generation/hashing/encryption methods in the right order. To keep control during the lengthy generation stage, this method starts an additional Thread and continues to process requests from the user.

3.3.3. AuxiliaryKeyCreator::generate_rsa_keys()

```
1 usage   ReadySetGet
def generate_rsa_keys(self, arg):
    keypair = RSA.generate(self._constants.LENGTH_OF_RSA_KEY)
    arg['key_priv'] = keypair.export_key(format=self._constants.KEY_FORMAT)
    arg['key_pub'] = keypair.public_key().export_key(format=self._constants.KEY_FORMAT)
```

Fig 5. AuxiliaryKeyCreator::generate_rsa_keys()

This method uses the pycryptodome library to generate the RSA keypair of the given length (4096 bits in this case). Afterwards, it exports both of the keys to the DER format.

3.3.4. DeviceListener::_on_message(int, int, int, int)

```
1 usage   ReadySetGet
def _on_message(self, hwnd: int, msg: int, wparam: int, lparam: int):
    if msg != win32con.WM_DEVICECHANGE:
        return 0
    event, description = self.WM_DEVICECHANGE_EVENTS[wparam]
    if event in ('DBT_DEVICEREMOVECOMPLETE', 'DBT_DEVICEARRIVAL'):
        self.on_change()
    return 0
```

Fig 6. DeviceListener::_on_message(int, int, int, int)

This method uses the Windows operating system's window mechanisms to intercept messages connected to attaching/detaching a pendrive ("DBT_DEVICEREMOVECOMPLETE" and "DBT_DEVICEARRIVAL" of the "WM_DEVICECHANGE" type). It then invokes the provided callback method (in this case - AuxiliaryGUI::on_devices_changed()).

3.3.5. AuxiliaryKeyCreator::gen_cert()

```
1 usage   new *
def gen_cert(self):

    timestamp_epoch_time_start = 0
    key = crypto.load_publickey(crypto.FILETYPE_PEM,
                                open("C:/Studia/BSK/ProjektBSK/AuxiliaryApp/ProjectBSKPublicKey.pem").read())
    keyPriv = crypto.load_privatekey(crypto.FILETYPE_PEM,
                                     open("D:/ProjectBSKPrivateKey.pem").read(), self._pin_hash.digest())
    timestamp_epoch_time_end = 10 * 365 * 24 * 60 * 60
    cert_sign = crypto.X509()
    cert_sign.get_subject().CN = "Adam Zarzycki 193243"
    cert_sign.set_serial_number(int(time.time()))
    cert_sign.gmtime_adj_notBefore(timestamp_epoch_time_start)
    cert_sign.gmtime_adj_notAfter(timestamp_epoch_time_end)
    cert_sign.set_issuer(cert_sign.get_subject())
    cert_sign.set_pubkey(key)
    cert_sign.add_extensions([crypto.X509Extension(b"keyUsage", False, b"digitalSignature,nonRepudiation")])
    cert_sign.sign(keyPriv, "sha256")
    sign_cert = crypto.dump_certificate(crypto.FILETYPE_PEM, cert_sign)

    with open("certyfikat.pem", "wb") as certfile:
        certfile.write(sign_cert)
```

Fig 7. AuxiliaryKeyCreator::gen_cert()

This method uses the pyOpenSSL cryptography library to generate a new self-signed X.509 certificate for the future signing process. It is called after writing the keys to their respective files, so it loads them back (while decrypting the private key with the passphrase provided in the third parameter). It sets the subject's name ("get_subject().CN"), validity period (to 10 years, variable "timestamp_epoch_time_end"), issuer and certified public key. It also sets the necessary KeyUsage extension to digital signatures and non-repudiation capabilities. Lastly, it saves the certificate in the "certyfikat.pem" file.

3.4. Tests summary

- key generation without the pendrive attached - impossible, the "Start generation" button is not enabled,
- non-numerical value given as PIN - the request is repeated until a numerical value is provided,
- no key/certificate files present - new files are created in the paths intended,
- old key/certificate files present - files are overwritten with newly-generated values.

3.5. Used technology

- operating system - Windows 10,
- programming language - Python 3.11,
- GUI - PySide6,
- generation/hashing/encryption - pycryptodome 3.21.0,
- certificate generation - pyopenssl 25.1.0,
- device recognition - win32api,
- threading - threading.Thread.

4. Solution

4.1. Short description of the Solution

I prepared the main app of the project, according to the requirements provided in the project instruction. Its functionalities are as follows:

- automatic pendrive/private key detection,
- automatic loading of the private key to the application,
- obtaining a numerical PIN from the user,
- checking correctness of the PIN provided,
- decrypting the private key with a SHA256 hash of the PIN provided,
- letting User A choose the PDF file to be signed,
- signing the file with a PAdES qualified signature,
- saving the signed file,
- letting User B choose the PDF file which signature is to be verified,
- verifying the given file with User A's public key,
- a GUI showing the Users stages of execution taking place, and the result of the verification process.

4.2. Simplified block diagram of the Solution

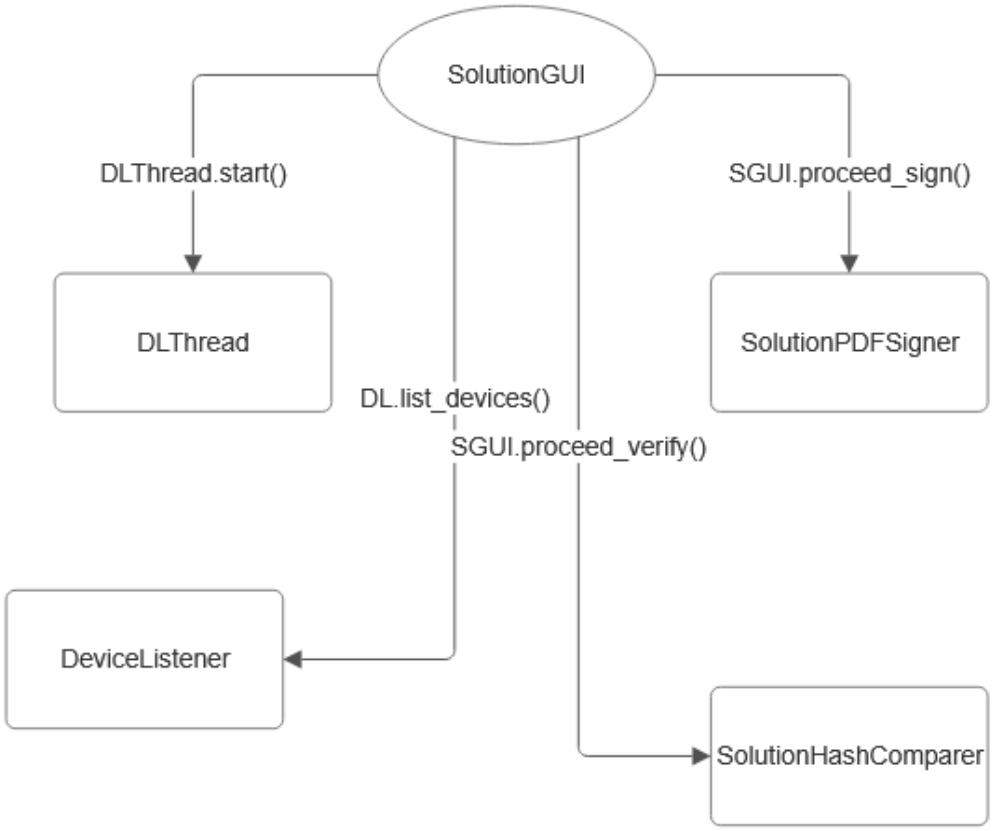

Fig 8. Simplified block diagram of the Solution

Explanation:

The Solution has a structure rather similar to the AuxiliaryApp. The SolutionGUI, DLThread, and DeviceListener have mostly the same behaviour as their auxiliary counterparts, with a few exceptions:

- SolutionGUI checks not only for a pendrive, but also for the presence of a private key on it,
- SolutionGUI automatically starts the signing process if a pendrive with a private key is detected, also on the start of the application.

The SolutionPDFSigner is invoked automatically by the SolutionGUI class to sign the chosen .pdf document (or on User's demand - as the signing process can also be invoked manually by them). The resulting PDF file with a signature is then saved in the place of its predecessor.

The SolutionHashComparer is invoked by the SolutionGUI manually, after receiving a command from the User. It lets them choose the PDF to be verified and returns to the SolutionGUI the result of this verification, which shows it to the User afterwards.

4.3. Description of the most important methods

4.3.1. SolutionGUI::proceed_verify()

```
self.show_current_arrow(self._current_stage_nr)
valid = self._hash_comparer.verify()
if valid == -1:
    self.generation_stages_init()
    self._button_sign.setEnabled(True if self._is_d_drive_connected else False)
    self._button_sign.repaint()

    self._button_verify.setEnabled(True)
    self._button_verify.repaint()
    self._result_comm.setText("Plik nie jest podpisany")
    return
self.set_texts_verify()
time.sleep(0.5)
```

Fig 9. SolutionGUI:proceed_verify() (fragment)

This method controls the flow of the verification process, by systematically calling the right methods of the “SolutionHashComparer” class. It works very similarly for every stage of this process, so on the listing only a logical fragment is shown.

First, it sets the current stage indicator by calling “show_current_arrow()” method. Then, it calls the “verify()” method of “SolutionHashComparer” class, which compares the signature with its supposed value. Then, it processes the return value, and changes the visible state of the application accordingly.

4.3.2. SolutionGUI::proceed_sign()

```
self.show_current_arrow(self._current_stage_nr)
was_good_pin = self._signer.decrypt()
if not was_good_pin:
    self.generation_stages_init()
    self._button_sign.setEnabled(True if self._is_d_drive_connected else False)
    self._button_sign.repaint()

    self._button_verify.setEnabled(True)
    self._button_verify.repaint()
    self._result_comm.setText("Niepoprawny klucz")
    return
self.set_texts_sign()
time.sleep(0.5)
```

Fig 10. SolutionGUI::proceed_sign() (fragment)

This method has the exact same structure as the "proceed_verify()" method described earlier, but it focuses on controlling the signing process flow instead. The fragment provided shows a call to "decrypt" method, which, besides decrypting the private key with the PIN hash, returns the result of a decryption check (whether it was the same PIN that was used for encrypting the key), so the GUI can adapt, and eventually stop the process.

4.3.3. SolutionPDFSigner::sign()

```
1 usage
def sign(self):

    with open(self._file_to_sign_path + self._file_to_sign, 'rb') as inf:
        w = IncrementalPdfFileWriter(inf, strict=False)
        with open('../pdfs/signed'+self._file_to_sign, 'wb') as outf:
            signers.sign_pdf(
                w, signature_meta=self._signature_meta, signer=self._cms_signer,
                output=outf
            )

    os.remove(self._file_to_sign_path + self._file_to_sign)
```

Fig 11. SolutionPDFSigner::sign()

This method uses the pyHanko digital signature library to sign the provided PDF file ("inf") to an output file ("outf"), using configuration from an earlier call to the "SolutionPDFSigner::prepare_file()" method. It also removes the original file.

4.3.4. SolutionHashComparer::verify()

```
1 usage
def verify(self):

    with open(self._file_path + self._file_name, 'rb') as doc:
        self._r = PdfFileReader(doc, strict=False)
        if len(self._r.embedded_signatures) == 0:
            return -1
        self._sig = self._r.embedded_signatures[0]
        status = validate_pdf_signature(self._sig, self._vc)
        print(status.pretty_print_details())
        if status.bottom_line:
            return 1

        return 0
```

Fig 12. SolutionHashComparer::verify()

This method opens the chosen signed PDF file and extracts its signature, after which it uses the pyHanko method “validate_pdf_signature()” to hash the provided file and compare it with the signature. Then, it checks the overall validity of the signature (“status.bottom_line”), and returns accordingly.

It is worth mentioning that if there is no signature in the chosen file, this method returns with a different value, so the “SolutionGUI” class can show an adequate message.

4.4. Testing

- pendrive attached on program start - automatically detected, key loaded,
- pendrive not attached on program start - signing impossible,
- pendrive attached later - automatically detected, key loaded,
- key not present on pendrive - signing impossible,
- non-numerical PIN - the request is repeated until a numerical value is provided,
- wrong PIN (not the one the key was encrypted with) - the process is stopped with an “Incorrect key” message,

- choosing of a non-PDF file - impossible, only PDF files are shown,
- no file chosen - the process is stopped with a “No file chosen” message,
- signing an already-signed PDF - the process is stopped with a “File is already signed” message,
- verifying a file without a signature - the process is stopped with a “File is not signed” message,
- **verifying a file with a correct signature -** the process ends with a “Verification succeeded - the document is correct”,
- **verifying a file that was changed after signing -** the process ends with a “Verification failed - document had been changed” message.

4.5. Used technology

- operating system - Windows 10,
- programming language - Python 3.11,
- GUI - PySide6,
- hashing/decryption - pycryptodome 3.23.0,
- signing/verifying/PDF writing/reading - pyhanko 0.29.0 / pyhanko-certvalidator 0.27.0,
- device recognition - win32api,
- threading - threading.Thread.

5. Link to Github repository

Link: https://github.com/ReadySetGet/Projekt-BSK

6. Literature

- the basic skeleton of DeviceListener class: https://abdus.dev/posts/python-monitor-usb/,
- pycryptodome online documentation: https://www.pycryptodome.org/,
- PySide6 online documentation: https://doc.qt.io/qtforpython-6/index.html,
- pyOpenSSL online documentation: https://www.pyopenssl.org/en/stable/index.html,
- pyHanko online documentation: https://pyhanko.readthedocs.io/en/latest/lib-guide/index.html,
- the instruction and requirements provided on the enauczanie platform.