Tool for Emulating the PAdES Qualified Electronic Signature

Generated by Doxygen 1.9.8

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 auxiliary_app Namespace Reference

Namespaces

- namespace gui
- namespace main
- namespace utils

5.1.1 Detailed Description

```
auxiliary_app

This module provides the auxiliary application for the electronic signature project. It includes functionality

Modules:

- gui
    - key_generator_window.py
        - KeyGeneratorWindow: A window for generating RSA keys with a graphical user interface.
            - Methods:
                - __init__(): Initializes the KeyGeneratorWindow instance and sets up the UI.
                - init_ui(): Sets up the user interface components, including buttons and layout.
                - open_pin_pad(): Opens a PIN pad dialog for the user to enter a PIN before generating keys.
                - start_key_generation(pin): Starts the key generation process in a separate thread and shows
                - update_progress(message, value): Updates the progress dialog with the current progress of th
                - handle_status(status_code, message): Handles the status updates from the key generation thre
                - close_application(): Closes the application when the quit button is clicked.

    - key_generation_thread.py
        - KeyGenerationThread: A QThread subclass responsible for generating RSA keys in a separate thread.
            - Signals:
                - progress_update (str, int): Emitted to update the progress of the RSA key generation process
                - status (RsaGenState, str): Emitted to indicate the status of the RSA key generation process.
            - Attributes:
                - pin (str): The PIN code used for RSA key generation.
                - drive_manager (DriveManager): The drive manager instance used for managing drives during RSA
            - Methods:
                - __init__(pin, drive_manager): Initializes the KeyGenerationThread instance with the provided
                - run(): Executes the RSA key generation process and emits progress and status updates.

    - enums.py
        - RsaGenState: Enum representing the state of RSA key generation.
            - Attributes:
                - FINISHED (int): Indicates that the RSA key generation has finished successfully.
                - ERRORED (int): Indicates that an error occurred during RSA key generation.
```

```
- utils
    - utils.py
        - generate_rsa_keys(pin, drive_manager, progress_signal=None): Generates RSA keys, encrypts the privat
            - Args:
                - pin (str): The PIN used to hash and encrypt the private key.
                - drive_manager (DriveManager): An object responsible for managing the USB drive operations.
                - progress_signal (object, optional): An optional signal object to emit progress updates.
            - Raises:
                - Exception: If any error occurs during the key generation process.
            - Emits:
                - progress_signal (str, int): Emits progress updates with a message and a percentage.
```

5.2 auxiliary_app.gui Namespace Reference

Namespaces

- namespace enums
- namespace key_generation_thread
- namespace key_generator_window

5.2.1 Detailed Description

```
auxiliary_app.gui

This module provides the graphical user interface (GUI) components for the auxiliary application. It includes

Modules:

- key_generator_window.py
    - KeyGeneratorWindow: A window for generating RSA keys with a graphical user interface.
        - Methods:
            - __init__(): Initializes the KeyGeneratorWindow instance and sets up the UI.
            - init_ui(): Sets up the user interface components, including buttons and layout.
            - open_pin_pad(): Opens a PIN pad dialog for the user to enter a PIN before generating keys.
            - start_key_generation(pin): Starts the key generation process in a separate thread and shows a pr
            - update_progress(message, value): Updates the progress dialog with the current progress of the ke
            - handle_status(status_code, message): Handles the status updates from the key generation thread,
            - close_application(): Closes the application when the quit button is clicked.

- key_generation_thread.py
    - KeyGenerationThread: A QThread subclass responsible for generating RSA keys in a separate thread.
        - Signals:
            - progress_update (str, int): Emitted to update the progress of the RSA key generation process.
            - status (RsaGenState, str): Emitted to indicate the status of the RSA key generation process.
        - Attributes:
            - pin (str): The PIN code used for RSA key generation.
            - drive_manager (DriveManager): The drive manager instance used for managing drives during RSA key
        - Methods:
            - __init__(pin, drive_manager): Initializes the KeyGenerationThread instance with the provided PIN
            - run(): Executes the RSA key generation process and emits progress and status updates.

- enums.py
    - RsaGenState: Enum representing the state of RSA key generation.
        - Attributes:
            - FINISHED (int): Indicates that the RSA key generation has finished successfully.
            - ERRORED (int): Indicates that an error occurred during RSA key generation.
```

5.3 auxiliary_app.gui.enums Namespace Reference

Classes

- class RsaGenState

5.4 auxiliary_app.gui.key_generation_thread Namespace Reference

Classes

- class KeyGenerationThread

Variables

- logger = logging.getLogger("global_logger")

5.4.1 Variable Documentation

5.4.1.1 logger

```
auxiliary_app.gui.key_generation_thread.logger = logging.getLogger("global_logger")
```

5.5 auxiliary_app.gui.key_generator_window Namespace Reference

Classes

- class KeyGeneratorWindow

Variables

- logger = logging.getLogger("global_logger")

5.5.1 Variable Documentation

5.5.1.1 logger

```
auxiliary_app.gui.key_generator_window.logger = logging.getLogger("global_logger")
```

5.6 auxiliary_app.main Namespace Reference

Variables

- logger = initialize(AUXILIARY_LOG_FILE)
- dev_manager = DriveManager()
- app = QApplication(sys.argv)
- window = KeyGeneratorWindow()

5.6.1 Variable Documentation

5.6.1.1 app

```
auxiliary_app.main.app = QApplication(sys.argv)
```

5.6.1.2 dev_manager

```
auxiliary_app.main.dev_manager = DriveManager()
```

5.6.1.3 logger

```
auxiliary_app.main.logger = initialize(AUXILIARY_LOG_FILE)
```

5.6.1.4 window

```
auxiliary_app.main.window = KeyGeneratorWindow()
```

5.7 auxiliary_app.utils Namespace Reference

Namespaces

- namespace utils

5.7.1 Detailed Description

```
auxiliary_app.utils

This module provides utility functions for the auxiliary application. It includes functions for generating RSA

Modules:

- utils.py
    - generate_rsa_keys(pin, drive_manager, progress_signal=None): Generates RSA keys, encrypts the private ke
        - Args:
            - pin (str): The PIN used to hash and encrypt the private key.
            - drive_manager (DriveManager): An object responsible for managing the USB drive operations.
            - progress_signal (object, optional): An optional signal object to emit progress updates.
        - Raises:
            - Exception: If any error occurs during the key generation process.
        - Emits:
            - progress_signal (str, int): Emits progress updates with a message and a percentage.
```

5.8 auxiliary_app.utils.utils Namespace Reference

Functions

- generate_rsa_keys (pin, drive_manager, progress_signal=None)

Variables

- logger = logging.getLogger("global_logger")

5.8.1 Function Documentation

5.8.1.1 generate_rsa_keys()

```
auxiliary_app.utils.utils.generate_rsa_keys (
              pin,
              drive_manager,
              progress_signal = None )
```

```
Generates RSA keys, encrypts the private key with a hashed PIN, and saves both keys to a USB drive.

Args:
    pin (str): The PIN used to hash and encrypt the private key.
    drive_manager (object): An object responsible for managing the USB drive operations.
    progress_signal (object, optional): An optional signal object to emit progress updates.

Raises:
    Exception: If any error occurs during the key generation process.

Emits:
    progress_signal (str, int): Emits progress updates with a message and a percentage.
```

5.8.2 Variable Documentation

5.8.2.1 logger

```
auxiliary_app.utils.utils.logger = logging.getLogger("global_logger")
```

5.9 common Namespace Reference

Namespaces

- namespace drive_manager
- namespace gui
- namespace logger
- namespace utils

5.9.1 Detailed Description

```
common

This module provides common functionalities for the electronic signature project. It includes submodules for m

Modules:

- drive_manager
    - drive_manager.py
        - DriveManager: A class responsible for managing USB drives.
            - Methods:
                - __init__(): Initializes the DriveManager instance.
                - refresh() -> list[str]: Refreshes and returns a list of USB drives.
                - list_drives_with_keys() -> list[str]: Returns a list of USB drives that contain specific key
                - read_files(path: str) -> list[str]: Reads and returns a list of filenames from the specifiec
                - save_to_drive(data: bytes, destination_name: str) -> bool: Saves binary data to a file on th

- gui
    - drive_selection.py
        - DriveSelectionWidget: A widget for selecting a drive from a list of connected drives.
            - Methods:
                - __init__(mode=DriveSelectorMode.STANDARD): Initializes the DriveSelectionWidget.
                - init_ui(): Initializes the user interface.
                - refresh_drives(): Refreshes the list of connected drives.
                - get_connected_drives(): Retrieves the list of connected drives based on the mode.
                - select_drive(): Selects the currently highlighted drive in the list.

    - pin_pad_dialog.py
        - PinPadDialog: A dialog window for entering a PIN code.
            - Methods:
                - __init__(): Initializes a new instance of the PinPad dialog.
                - init_ui(): Initializes the user interface for the PIN pad dialog.
                - add_number(number): Adds a number to the current PIN.
                - clear_pin(): Clears the current PIN.
                - backspace(): Removes the last digit from the current PIN.
                - get_pin(): Returns the current PIN.

    - enums.py
        - DriveSelectorMode: Enumeration for drive selector modes.
            - Attributes:
                - STANDARD (int): Standard drive selection mode.
                - WITH_KEYS (int): Drive selection mode with keys.

- logger
    - logger.py
        - compress_old_log(log_file): Compresses the existing log file into a single ZIP archive before starti
            - Args:
                - log_file (Path): The path to the log file to be compressed.
        - initialize(log_file): Initializes the new global logger instance.
            - Args:
                - log_file (Path): The path to the log file to be initialized.

- utils
    - utils.py
        - load_stylesheet(widget, relative_path): Loads a stylesheet from a given relative path and applies it
            - Args:
                - widget (QWidget): The widget to which the stylesheet will be applied.
                - relative_path (str): The relative path to the stylesheet file.
            - Raises:
                - FileNotFoundError: If the stylesheet file does not exist.
                - IOError: If there is an error reading the stylesheet file.
```

5.10 common.drive_manager Namespace Reference

Namespaces

- namespace drive_manager

5.10.1 Detailed Description

```
common.drive_manager

This module provides functionality for managing USB drives. It includes classes and methods for listing availa

Modules:

- drive_manager.py
    - DriveManager: A class responsible for managing USB drives.
        - Methods:
            - __init__(): Initializes the DriveManager instance.
            - refresh() -> list[str]: Refreshes and returns a list of USB drives.
            - list_drives_with_keys() -> list[str]: Returns a list of USB drives that contain specific key fil
            - read_files(path: str) -> list[str]: Reads and returns a list of filenames from the specified dis
            - save_to_drive(data: bytes, destination_name: str) -> bool: Saves binary data to a file on the se
```

5.11 common.drive_manager.drive_manager Namespace Reference

Classes

- class DriveManager

Variables

- logger = logging.getLogger("global_logger")

5.11.1 Variable Documentation

5.11.1.1 logger

```
common.drive_manager.drive_manager.logger = logging.getLogger("global_logger")
```

5.12 common.gui Namespace Reference

Namespaces

- namespace drive_selection
- namespace enums
- namespace pin_pad_dialog

5.12.1 Detailed Description

```
common.gui

This module provides graphical user interface (GUI) components for the common functionalities of the electroni

Modules:

- drive_selection.py
    - DriveSelectionWidget: A widget for selecting a drive from a list of connected drives.
        - Methods:
            - __init__(mode=DriveSelectorMode.STANDARD): Initializes the DriveSelectionWidget.
            - init_ui(): Initializes the user interface.
            - refresh_drives(): Refreshes the list of connected drives.
            - get_connected_drives(): Retrieves the list of connected drives based on the mode.
            - select_drive(): Selects the currently highlighted drive in the list.

- pin_pad_dialog.py
    - PinPadDialog: A dialog window for entering a PIN code.
        - Methods:
            - __init__(): Initializes a new instance of the PinPad dialog.
            - init_ui(): Initializes the user interface for the PIN pad dialog.
            - add_number(number): Adds a number to the current PIN.
            - clear_pin(): Clears the current PIN.
            - backspace(): Removes the last digit from the current PIN.
            - get_pin(): Returns the current PIN.

- enums.py
    - DriveSelectorMode: Enumeration for drive selector modes.
        - Attributes:
            - STANDARD (int): Standard drive selection mode.
            - WITH_KEYS (int): Drive selection mode with keys.
```

5.13 common.gui.drive_selection Namespace Reference

Classes

- class DriveSelectionWidget

Variables

- logger = logging.getLogger("global_logger")
- int DRIVES_REFRESH = 300

5.13.1 Variable Documentation

5.13.1.1 DRIVES_REFRESH

```
int common.gui.drive_selection.DRIVES_REFRESH = 300
```

5.13.1.2 logger

```
common.gui.drive_selection.logger = logging.getLogger("global_logger")
```

5.14 common.gui.enums Namespace Reference

Classes

- class DriveSelectorMode

5.15 common.gui.pin_pad_dialog Namespace Reference

Classes

- class PinPadDialog

Variables

- logger = logging.getLogger("global_logger")

5.15.1 Variable Documentation

5.15.1.1 logger

```
common.gui.pin_pad_dialog.logger = logging.getLogger("global_logger")
```

5.16 common.logger Namespace Reference

Namespaces

- namespace logger

5.16.1 Detailed Description

```
common.logger

This module provides logging functionality for the electronic signature project. It includes functions for ini

Modules:

- logger.py
    - compress_old_log(log_file): Compresses the existing log file into a single ZIP archive before starting a
        - Args:
            - log_file (Path): The path to the log file to be compressed.
    - initialize(log_file): Initializes the new global logger instance.
        - Args:
            - log_file (Path): The path to the log file to be initialized.
```

5.17 common.logger.logger Namespace Reference

Functions

- compress_old_log (log_file)
- initialize (log_file)

Variables

- AUXILIARY_LOG_FILE = Path("auxiliary.log")
- MAIN_LOG_FILE = Path("main.log")
- ZIP_FILE = Path("logs.zip")

5.17.1 Function Documentation

5.17.1.1 compress_old_log()

```
common.logger.logger.compress_old_log (
            log_file )
```

Compresses the existing log file into a single ZIP archive before starting a new session

5.17.1.2 initialize()

```
common.logger.logger.initialize (
            log_file )
```

Initializes the new global logger instance

5.17.2 Variable Documentation

5.17.2.1 AUXILIARY_LOG_FILE

```
common.logger.logger.AUXILIARY_LOG_FILE = Path("auxiliary.log")
```

5.17.2.2 MAIN_LOG_FILE

```
common.logger.logger.MAIN_LOG_FILE = Path("main.log")
```

5.17.2.3 ZIP_FILE

```
common.logger.logger.ZIP_FILE = Path("logs.zip")
```

5.18 common.utils Namespace Reference

Namespaces

- namespace utils

5.18.1 Detailed Description

```
common.utils

This module provides utility functions for the electronic signature project. It includes functions for loading

Modules:

- utils.py
    - load_stylesheet(widget, relative_path): Loads a stylesheet from a given relative path and applies it to
        - Args:
            - widget (QWidget): The widget to which the stylesheet will be applied.
            - relative_path (str): The relative path to the stylesheet file.
        - Raises:
            - FileNotFoundError: If the stylesheet file does not exist.
            - IOError: If there is an error reading the stylesheet file.
```

5.19 common.utils.utils Namespace Reference

Functions

- load_stylesheet (widget, relative_path)

Variables

- logger = logging.getLogger("global_logger")

5.19.1 Function Documentation

5.19.1.1 load_stylesheet()

```
common.utils.utils.load_stylesheet (
              widget,
              relative_path )
```

```
Loads a stylesheet from a given relative path and applies it to the specified widget.

Args:
    widget (QWidget): The widget to which the stylesheet will be applied.
    relative_path (str): The relative path to the stylesheet file.

Raises:
    FileNotFoundError: If the stylesheet file does not exist.
    IOError: If there is an error reading the stylesheet file.
```

5.19.2 Variable Documentation

5.19.2.1 logger

```
common.utils.utils.logger = logging.getLogger("global_logger")
```

5.20 main_app Namespace Reference

Namespaces

- namespace gui
- namespace main
- namespace utils

5.20.1 Detailed Description

```
main_app

This module provides the main application for the electronic signature project. It includes functionality for

Modules:

- gui
    - sign_and_verify.py
        - SignVerifyWindow: A window for signing and verifying PDF files.
            - Methods:
                - __init__(): Initializes the SignVerifyWindow instance and sets up the UI.
                - init_ui(): Sets up the user interface for the window, including buttons for signing, verifyi
                - start_signing_file(pin, pdf_path): Starts the process of signing a PDF file, showing a progr
                - start_verifying_file(pub_key_path, pdf_path): Starts the process of verifying a PDF file, sh
                - update_progress(message, value): Updates the progress dialog with the current progress messa
                - handle_status(status_code, message): Handles the status updates from the signing or verifyin
                - verify_sign(): Initiates the process of verifying a PDF file by selecting the PDF and public
                - sign_pdf(): Initiates the process of signing a PDF file by opening a PIN dialog, selecting t
                - select_pdf_file(): Opens a file dialog to select a PDF file for signing or verifying.
                - select_pub_key_file(): Opens a file dialog to select a public key file for verifying a PDF.
                - close_application(): Closes the application and logs the closure.

    - sign_thread.py
        - SignThread: A QThread subclass to handle the process of signing a PDF file in a separate thread.
            - Signals:
                - progress_update (str, int): Emitted to update the progress of the signing process.
                - status (SignState, str): Emitted to update the status of the signing process.
            - Attributes:
                - pin (str): The PIN code used for RSA key decryption.
                - drive_manager (DriveManager): The drive manager instance to manage the drive operations.
                - pdf_path (str): The file path of the PDF to be signed.
            - Methods:
                - __init__(pin, drive_manager, pdf_path): Initializes the SignThread class with the provided P
                - run(): Executes the signing process, emitting progress updates and status changes.

    - verify_thread.py
        - VerifyThread: A QThread subclass to handle the verification of a PDF file in a separate thread.
            - Signals:
                - progress_update (str, int): Emitted to update the progress of the verification process.
                - status (VerifyState, str): Emitted to update the status of the verification process.
            - Attributes:
                - pub_key_path (str): The file path to the public key used for verification.
                - pdf_path (str): The file path to the PDF file to be verified.
            - Methods:
                - __init__(pub_key_path, pdf_path): Initializes the VerifyThread instance with the provided pu
                - run(): Executes the verification process, emitting progress updates and status changes.
```

```
    - enums.py
        - SignState: Enumeration representing the state of a signing process.
            - Attributes:
                - FINISHED (int): Indicates that the signing process has completed successfully.
                - ERRORED (int): Indicates that an error occurred during the signing process.
        - VerifyState: Enumeration representing the state of a verification process.
            - Attributes:
                - FINISHED (int): Indicates that the verification process has finished successfully.
                - ERRORED (int): Indicates that an error occurred during the verification process.

- utils
    - pdf_utils.py
        - sign_pdf(pdf_path, rsa_key, progress_signal=None): Signs a PDF file using the provided RSA key.
            - Args:
                - pdf_path (str): The path to the PDF file to be signed.
                - rsa_key (RSA.RsaKey): The RSA key to use for signing the PDF.
                - progress_signal (optional): A signal to report progress, if applicable.
            - Raises:
                - Exception: If an error occurs during the signing process.
        - verify_pdf(pdf_path, public_key, progress_signal=None) -> bool: Verifies the digital signature of a
            - Args:
                - pdf_path (str): The file path to the PDF document to be verified.
                - public_key (RSA.RsaKey): The public RSA key used to verify the signature.
                - progress_signal (optional): A signal to report progress, if applicable.
            - Returns:
                - bool: True if the PDF signature is valid, False otherwise.
            - Raises:
                - Exception: If an error occurs during the verification process.

    - crypto_utils.py
        - read_public_key(public_key_path) -> RSA.RsaKey: Reads an RSA public key from the specified file path
            - Args:
                - public_key_path (str or Path): The path to the public key file.
            - Returns:
                - RSA.RsaKey: The RSA public key.
            - Raises:
                - ValueError: If the key is invalid or corrupted.
                - KeyError: If the key is invalid or corrupted.
                - FileNotFoundError: If the specified file does not exist.
                - Exception: For any other unexpected errors during key decryption.
        - decrypt_rsa_key(pin, drive_manager, progress_signal=None) -> RSA.RsaKey: Decrypts an RSA private key
            - Args:
                - pin (str): The PIN used to decrypt the RSA key.
                - drive_manager: An object that manages the drive where the encrypted key is stored.
                - progress_signal (optional): A signal object to emit progress updates. Defaults to None.
            - Returns:
                - RSA.RsaKey: The decrypted RSA private key.
            - Raises:
                - Exception: If the decryption fails due to an invalid PIN, corrupted key, file not found, or
```

5.21 main_app.gui Namespace Reference

Namespaces

- namespace enums
- namespace sign_and_verify
- namespace sign_thread
- namespace verify_thread

5.21.1 Detailed Description

main_app.gui

```
This module provides graphical user interface (GUI) components for the main application of the electronic sign

Modules:

- sign_and_verify.py
    - SignVerifyWindow: A window for signing and verifying PDF files.
        - Methods:
            - __init__(): Initializes the SignVerifyWindow instance and sets up the UI.
            - init_ui(): Sets up the user interface for the window, including buttons for signing, verifying,
            - start_signing_file(pin, pdf_path): Starts the process of signing a PDF file, showing a progress
            - start_verifying_file(pub_key_path, pdf_path): Starts the process of verifying a PDF file, showin
            - update_progress(message, value): Updates the progress dialog with the current progress message a
            - handle_status(status_code, message): Handles the status updates from the signing or verifying pr
            - verify_sign(): Initiates the process of verifying a PDF file by selecting the PDF and public key
            - sign_pdf(): Initiates the process of signing a PDF file by opening a PIN dialog, selecting the P
            - select_pdf_file(): Opens a file dialog to select a PDF file for signing or verifying.
            - select_pub_key_file(): Opens a file dialog to select a public key file for verifying a PDF.
            - close_application(): Closes the application and logs the closure.

- sign_thread.py
    - SignThread: A QThread subclass to handle the process of signing a PDF file in a separate thread.
        - Signals:
            - progress_update (str, int): Emitted to update the progress of the signing process.
            - status (SignState, str): Emitted to update the status of the signing process.
        - Attributes:
            - pin (str): The PIN code used for RSA key decryption.
            - drive_manager (DriveManager): The drive manager instance to manage the drive operations.
            - pdf_path (str): The file path of the PDF to be signed.
        - Methods:
            - __init__(pin, drive_manager, pdf_path): Initializes the SignThread class with the provided PIN,
            - run(): Executes the signing process, emitting progress updates and status changes.

- verify_thread.py
    - VerifyThread: A QThread subclass to handle the verification of a PDF file in a separate thread.
        - Signals:
            - progress_update (str, int): Emitted to update the progress of the verification process.
            - status (VerifyState, str): Emitted to update the status of the verification process.
        - Attributes:
            - pub_key_path (str): The file path to the public key used for verification.
            - pdf_path (str): The file path to the PDF file to be verified.
        - Methods:
            - __init__(pub_key_path, pdf_path): Initializes the VerifyThread instance with the provided public
            - run(): Executes the verification process, emitting progress updates and status changes.

- enums.py
    - SignState: Enumeration representing the state of a signing process.
        - Attributes:
            - FINISHED (int): Indicates that the signing process has completed successfully.
            - ERRORED (int): Indicates that an error occurred during the signing process.
    - VerifyState: Enumeration representing the state of a verification process.
        - Attributes:
            - FINISHED (int): Indicates that the verification process has finished successfully.
            - ERRORED (int): Indicates that an error occurred during the verification process.
```

5.22 main_app.gui.enums Namespace Reference

Classes

- class SignState
- class VerifyState

5.23 main_app.gui.sign_and_verify Namespace Reference

Classes

- class SignVerifyWindow

Variables

- logger = logging.getLogger("global_logger")

5.23.1 Variable Documentation

5.23.1.1 logger

```
main_app.gui.sign_and_verify.logger = logging.getLogger("global_logger")
```

5.24 main_app.gui.sign_thread Namespace Reference

Classes

- class SignThread

Variables

- logger = logging.getLogger("global_logger")

5.24.1 Variable Documentation

5.24.1.1 logger

```
main_app.gui.sign_thread.logger = logging.getLogger("global_logger")
```

5.25 main_app.gui.verify_thread Namespace Reference

Classes

- class VerifyThread

Variables

- logger = logging.getLogger("global_logger")

5.25.1 Variable Documentation

5.25.1.1 logger

```
main_app.gui.verify_thread.logger = logging.getLogger("global_logger")
```

5.26 main_app.main Namespace Reference

Variables

- logger = initialize(MAIN_LOG_FILE)
- dev_manager = DriveManager()
- app = QApplication(sys.argv)
- window = SignVerifyWindow()

5.26.1 Variable Documentation

5.26.1.1 app

```
main_app.main.app = QApplication(sys.argv)
```

5.26.1.2 dev_manager

```
main_app.main.dev_manager = DriveManager()
```

5.26.1.3 logger

```
main_app.main.logger = initialize(MAIN_LOG_FILE)
```

5.26.1.4 window

```
main_app.main.window = SignVerifyWindow()
```

5.27 main_app.utils Namespace Reference

Namespaces

- namespace crypto_utils
- namespace pdf_utils

5.28 main_app.utils.crypto_utils Namespace Reference

Functions

- RSA.RsaKey read_public_key (public_key_path)
- RSA.RsaKey decrypt_rsa_key (str pin, drive_manager, progress_signal=None)

Variables

- logger = logging.getLogger("global_logger")

5.28.1 Function Documentation

5.28.1.1 decrypt_rsa_key()

```
RSA.RsaKey main_app.utils.crypto_utils.decrypt_rsa_key (
             str pin,
              drive_manager,
              progress_signal = None )
```

```
Decrypts an RSA private key using a provided PIN and drive manager.

Args:
    pin (str): The PIN used to decrypt the RSA key.
    drive_manager: An object that manages the drive where the encrypted key is stored.
    progress_signal (optional): A signal object to emit progress updates. Defaults to None.

Returns:
    RSA.RsaKey: The decrypted RSA private key.

Raises:
    Exception: If the decryption fails due to an invalid PIN, corrupted key, file not found, or any other unex
```

5.28.1.2 read_public_key()

```
RSA.RsaKey main_app.utils.crypto_utils.read_public_key (
              public_key_path )
```

```
Reads an RSA public key from the specified file path.

Args:
    public_key_path (str or Path): The path to the public key file.

Returns:
    RSA.RsaKey: The RSA public key.

Raises:
    ValueError: If the key is invalid or corrupted.
    KeyError: If the key is invalid or corrupted.
    FileNotFoundError: If the specified file does not exist.
    Exception: For any other unexpected errors during key decryption.
```

5.28.2 Variable Documentation

5.28.2.1 logger

```
main_app.utils.crypto_utils.logger = logging.getLogger("global_logger")
```

5.29 main_app.utils.pdf_utils Namespace Reference

Functions

- sign_pdf (str pdf_path, RSA.RsaKey rsa_key, progress_signal=None)
- bool verify_pdf (str pdf_path, RSA.RsaKey public_key, progress_signal=None)
- check_pdf_exists (str pdf_path, progress_signal=None)
- initialize_signing_process (str pdf_path, progress_signal=None)
- read_pdf_file (str pdf_path)
- initialize_pdf_writer (str pdf_path)
- hash_pdf (bytes pdf_content, progress_signal=None)
- create_signature (RSA.RsaKey rsa_key, pdf_hash, progress_signal=None)
- add_signature_to_pdf (writer, reader, bytes signature, progress_signal=None)
- save_signed_pdf (str pdf_path, writer, progress_signal=None)
- read_pdf_metadata (str pdf_path, progress_signal=None)
- prepare_unsigned_pdf (reader, str pdf_path, progress_signal=None)
- verify_signature (RSA.RsaKey public_key, pdf_hash, bytes signature, str pdf_path, progress_signal=None)

Variables

- logger = logging.getLogger("global_logger")

5.29.1 Function Documentation

5.29.1.1 add_signature_to_pdf()

```
main_app.utils.pdf_utils.add_signature_to_pdf (
               writer,
               reader,
              bytes signature,
               progress_signal = None )

Adds a digital signature to a PDF file.

Args:
writer (PdfWriter): The PdfWriter object used to write the PDF.
reader (PdfReader): The PdfReader object used to read the PDF.
signature (bytes): The digital signature to be added to the PDF.
progress_signal (Signal, optional): A signal object to emit progress updates. Defaults to None.

Returns:
None
```

5.29.1.2 check_pdf_exists()

```
main_app.utils.pdf_utils.check_pdf_exists (
             str pdf_path,
              progress_signal = None )

Checks if a PDF file exists at the given path.

Args:
    pdf_path (str): The path to the PDF file.
    progress_signal (optional): A signal to emit progress updates.
                                If provided, emits an error message with 100% progress if the file is not foun

Raises:
    FileNotFoundError: If the PDF file does not exist at the specified path.
```

5.29.1.3 create_signature()

```
main_app.utils.pdf_utils.create_signature (
             RSA.RsaKey rsa_key,
              pdf_hash,
              progress_signal = None )
```

```
Creates a digital signature for a given PDF hash using the provided RSA key.

Args:
    rsa_key (RSA.RsaKey): The RSA key to sign the PDF hash.
    pdf_hash: The hash of the PDF to be signed.
    progress_signal (optional): A signal to emit progress updates. Defaults to None.

Returns:
    bytes: The digital signature of the PDF hash.
```

5.29.1.4 hash_pdf()

```
main_app.utils.pdf_utils.hash_pdf (
             bytes pdf_content,
              progress_signal = None )
```

```
Hashes the content of a PDF file using SHA-256.

Args:
    pdf_content (bytes): The content of the PDF file to be hashed.
    progress_signal (optional): A signal to emit progress updates.
                                If provided, it will emit a message indicating the progress of the hashing pro

Returns:
    SHA256: The SHA-256 hash object of the PDF content.
```

5.29.1.5 initialize_pdf_writer()

```
main_app.utils.pdf_utils.initialize_pdf_writer (
             str pdf_path )
```

```
Initializes a PDF writer and reader for the given PDF file path.
This function reads the PDF file from the specified path and creates a PdfReader
and PdfWriter object. If an existing signature is found in the PDF metadata, it
removes the old signature.

Args:
    pdf_path (str): The file path to the PDF document.

Returns:
    tuple: A tuple containing the PdfReader and PdfWriter objects.
```

5.29.1.6 initialize_signing_process()

```
main_app.utils.pdf_utils.initialize_signing_process (
             str pdf_path,
              progress_signal = None )
```

```
Initializes the process of signing a PDF file.

Args:
    pdf_path (str): The path to the PDF file that needs to be signed.
    progress_signal (optional): A signal object to emit progress updates.
                                If provided, it should have an `emit` method
                                that accepts a message and a progress percentage.

Returns:
    None
```

5.29.1.7 prepare_unsigned_pdf()

```
main_app.utils.pdf_utils.prepare_unsigned_pdf (
              reader,
             str pdf_path,
              progress_signal = None )
```

```
Prepares an unsigned PDF by removing the signature metadata and returning the SHA256 hash of the PDF content.

Args:
    reader (PdfReader): The PDF reader object containing the PDF to be processed.
    pdf_path (str): The file path of the original PDF.
    progress_signal (Signal, optional): A signal object to emit progress updates. Defaults to None.

Returns:
    SHA256: The SHA256 hash of the unsigned PDF content.

Raises:
    Exception: If there is an error processing the PDF file.
```

5.29.1.8 read_pdf_file()

```
main_app.utils.pdf_utils.read_pdf_file (
             str pdf_path )
```

```
Reads the content of a PDF file.

Args:
    pdf_path (str): The path to the PDF file.

Returns:
    bytes: The content of the PDF file as bytes.
```

5.29.1.9 read_pdf_metadata()

```
main_app.utils.pdf_utils.read_pdf_metadata (
            str pdf_path,
             progress_signal = None )
```

```
Reads the metadata of a PDF file to extract the signature.

Args:
    pdf_path (str): The path to the PDF file.
    progress_signal (optional): A signal to emit progress updates. Defaults to None.

Returns:
    tuple: A tuple containing the PdfReader object and the signature in bytes.

Raises:
    ValueError: If no signature is found in the PDF metadata.
    Exception: If there is an error reading the PDF metadata.
```

5.29.1.10 save_signed_pdf()

```
main_app.utils.pdf_utils.save_signed_pdf (
            str pdf_path,
             writer,
             progress_signal = None )
```

```
Save a signed PDF file to the specified path.

Args:
    pdf_path (str): The path to the original PDF file.
    writer: The PDF writer object used to write the signed PDF.
    progress_signal (optional): A signal object to emit progress updates.

Returns:
    None
```

5.29.1.11 sign_pdf()

```
main_app.utils.pdf_utils.sign_pdf (
            str pdf_path,
            RSA.RsaKey rsa_key,
             progress_signal = None )
```

```
Signs a PDF file using the provided RSA key.

Args:
    pdf_path (str): The path to the PDF file to be signed.
    rsa_key (RSA.RsaKey): The RSA key to use for signing the PDF.
    progress_signal (optional): A signal to report progress, if applicable.

Raises:
    Exception: If an error occurs during the signing process.

This function performs the following steps:
    1. Checks if the PDF file exists.
    2. Initializes the signing process.
    3. Reads the content of the PDF file.
    4. Initializes the PDF writer and reader.
    5. Hashes the PDF content.
    6. Creates a signature using the RSA key and the PDF hash.
    7. Adds the signature to the PDF.
    8. Saves the signed PDF file.
```

5.29.1.12 verify_pdf()

```
bool main_app.utils.pdf_utils.verify_pdf (
             str pdf_path,
             RSA.RsaKey public_key,
              progress_signal = None )
```

```
Verifies the digital signature of a PDF file.

Args:
    pdf_path (str): The file path to the PDF document to be verified.
    public_key (RSA.RsaKey): The public RSA key used to verify the signature.
    progress_signal (optional): A signal to report progress, if applicable.

Returns:
    bool: True if the PDF signature is valid, False otherwise.

Raises:
    Exception: If an error occurs during the verification process.
```

5.29.1.13 verify_signature()

```
main_app.utils.pdf_utils.verify_signature (
             RSA.RsaKey public_key,
              pdf_hash,
             bytes signature,
             str pdf_path,
              progress_signal = None )
```

```
Verifies the digital signature of a PDF document.

Args:
    public_key (RSA.RsaKey): The RSA public key used to verify the signature.
    pdf_hash: The hash of the PDF document.
    signature (bytes): The digital signature to be verified.
    pdf_path (str): The file path of the PDF document.
    progress_signal (optional): A signal to emit progress updates.

Raises:
    ValueError: If the signature verification fails.
Emits:
    progress_signal: Emits progress updates if provided.
```

5.29.2 Variable Documentation

5.29.2.1 logger

```
main_app.utils.pdf_utils.logger = logging.getLogger("global_logger")
```

Chapter 6

Class Documentation

6.1 common.drive_manager.drive_manager.DriveManager Class Reference

Public Member Functions

- __init__ (self)
- list[str] refresh (self)
- list[str] list_drives_with_keys (self)
- list[str] read_files (self, str path)
- bool save_to_drive (self, bytes data, str destination_name)

Public Attributes

- drive_list
- selected_drive

6.1.1 Detailed Description

```
DriveManager is a class responsible for managing USB drives. It provides functionalities to list available dri
detect drives with specific key files, read files from a drive, and save data to a selected drive.

Methods:
    __init__():
    refresh() -> list[str]:
        Refreshes and returns a list of USB drives.
    list_drives_with_keys() -> list[str]:
        Returns a list of USB drives that contain specific key files.
    read_files(path: str) -> list[str]:
        Reads and returns a list of filenames from the specified disk path.
    save_to_drive(data: bytes, destination_name: str) -> bool:
```

6.1.2 Constructor & Destructor Documentation

6.1.2.1 __init__()

```
common.drive_manager.drive_manager.DriveManager.__init__ (
              self )
```

```
Initializes the DriveManager instance.
```

6.1.3 Member Function Documentation

6.1.3.1 list_drives_with_keys()

```
list[str] common.drive_manager.drive_manager.DriveManager.list_drives_with_keys (
              self )
```

```
Returns:
    list[str]: A list of USB drivers with key files
```

6.1.3.2 read_files()

```
list[str] common.drive_manager.drive_manager.DriveManager.read_files (
              self,
             str path )
```

```
Reads all files from the specified disk path.

Args:
    path (str): The path to the disk or directory.

Returns:
    list[str]: A list of filenames in the specified directory.
```

6.1.3.3 refresh()

```
list[str] common.drive_manager.drive_manager.DriveManager.refresh (
              self )
```

```
Returns:
    list[str]: A list of USB drivers
```

6.1.3.4 save_to_drive()

```
bool common.drive_manager.drive_manager.DriveManager.save_to_drive (
              self,
             bytes data,
             str destination_name )
```

```
Saves binary data to a file on the selected drive.

Args:
    data (bytes): Binary data to be saved on the USB drive.
    destination_name (str): Name of the file on the selected drive.

Returns:
    bool: True if the data is successfully saved, False otherwise.
```

6.1.4 Member Data Documentation

6.1.4.1 drive_list

`common.drive_manager.drive_manager.DriveManager.drive_list`

6.1.4.2 selected_drive

`common.drive_manager.drive_manager.DriveManager.selected_drive`

The documentation for this class was generated from the following file:

- common/drive_manager/drive_manager.py

6.2 common.gui.drive_selection.DriveSelectionWidget Class Reference

Inheritance diagram for common.gui.drive_selection.DriveSelectionWidget:

Collaboration diagram for common.gui.drive_selection.DriveSelectionWidget:

Public Member Functions

- __init__ (self, mode=DriveSelectorMode.STANDARD)
- init_ui (self)
- refresh_drives (self)
- get_connected_drives (self)
- select_drive (self)

Public Attributes

- mode
- drive_manager
- is_drive_selected
- selected_drive_label
- drive_list
- select_btn
- select_drive
- timer
- refresh_drives

6.2.1 Detailed Description

```
    DriveSelectionWidget is a QWidget that allows users to select a drive from a list of connected drives.

Attributes:
    mode (DriveSelectorMode): Mode of the drive selector, either 'STANDARD' or 'WITH_KEYS'.
    drive_manager (DriveManager): Manages the drives.
    is_drive_selected (bool): Indicates if a drive has been selected.
    selected_drive_label (QLabel): Label displaying the selected drive.
    drive_list (QListWidget): List widget displaying the available drives.
    select_btn (QPushButton): Button to select a drive.
    timer (QTimer): Timer to refresh the list of drives.

Methods:
    __init__(mode=DriveSelectorMode.STANDARD): Initializes the DriveSelectionWidget.
    init_ui(): Initializes the user interface.
    refresh_drives(): Refreshes the list of connected drives.
    get_connected_drives(): Retrieves the list of connected drives based on the mode.
    select_drive(): Selects the currently highlighted drive in the list.
```

6.2.2 Constructor & Destructor Documentation

6.2.2.1 __init__()

```
common.gui.drive_selection.DriveSelectionWidget.__init__ (
              self,
              mode = DriveSelectorMode.STANDARD )
```

```
DriveSelectionWidget constructor.

Args:
    mode (str): 'STANDARD' - lists all drives, 'WITH_KEYS' - only drives with keys.
```

6.2.3 Member Function Documentation

6.2.3.1 get_connected_drives()

```
common.gui.drive_selection.DriveSelectionWidget.get_connected_drives (
              self )
```

```
Retrieves a list of connected drives based on the current mode.
If the mode is `DriveSelectorMode.WITH_KEYS`, it lists drives that have keys.
Otherwise, it refreshes the drive manager and retrieves the full list of drives.

Returns:
    list: A list of connected drives.
```

6.2.3.2 init_ui()

```
common.gui.drive_selection.DriveSelectionWidget.init_ui (
              self )
```

```
Initializes the user interface for drive selection.
This method sets up the layout and widgets for the drive selection UI, including:
- Loading the stylesheet for the UI.
- Creating and configuring a vertical layout.
- Adding a label to display the selected drive.
- Adding a list widget to display available drives.
- Adding a button to confirm drive selection.
- Setting up a timer to periodically refresh the list of available drives.
Widgets:
    selected_drive_label (QLabel): Displays the currently selected drive.
    drive_list (QListWidget): Lists available drives for selection.
    select_btn (QPushButton): Button to confirm the selected drive.
    timer (QTimer): Timer to refresh the list of available drives.
Layouts:
    layout (QVBoxLayout): Main vertical layout for the UI.
    button_layout (QHBoxLayout): Horizontal layout for the select button.
Connections:
    select_btn.clicked: Connects to the select_drive method.
    timer.timeout: Connects to the refresh_drives method.
```

6.2.3.3 refresh_drives()

```
common.gui.drive_selection.DriveSelectionWidget.refresh_drives (
              self )
```

```
Refresh the list of connected drives in the GUI.
This method updates the drive list by performing the following steps:
1. Retrieves the currently connected drives.
2. Adds any new drives to the drive list.
3. Removes any drives from the drive list that are no longer connected.
4. If there is only one drive in the list and no drive is currently selected,
   it selects the first drive and marks it as selected.
5. Logs the refresh action.

Returns:
    None
```

6.2.3.4 select_drive()

```
common.gui.drive_selection.DriveSelectionWidget.select_drive (
              self )
```

```
Handles the selection of a drive from the drive list.

This method retrieves the selected item from the drive list. If an item is selected,
it updates the 'selected_drive' attribute of the 'drive_manager' with the text of the selected item,
logs the selected drive, and updates the 'selected_drive_label' to display the selected drive.
If no item is selected, it logs that no drive was selected and updates the 'selected_drive_label'
to indicate that no drive was selected.
```

6.2.4 Member Data Documentation

6.2.4.1 drive_list

common.gui.drive_selection.DriveSelectionWidget.drive_list

6.2.4.2 drive_manager

common.gui.drive_selection.DriveSelectionWidget.drive_manager

6.2.4.3 is_drive_selected

common.gui.drive_selection.DriveSelectionWidget.is_drive_selected

6.2.4.4 mode

common.gui.drive_selection.DriveSelectionWidget.mode

6.2.4.5 refresh_drives

common.gui.drive_selection.DriveSelectionWidget.refresh_drives

6.2.4.6 select_btn

common.gui.drive_selection.DriveSelectionWidget.select_btn

6.2.4.7 select_drive

common.gui.drive_selection.DriveSelectionWidget.select_drive

6.2.4.8 selected_drive_label

common.gui.drive_selection.DriveSelectionWidget.selected_drive_label

6.2.4.9 timer

common.gui.drive_selection.DriveSelectionWidget.timer

The documentation for this class was generated from the following file:

- common/gui/drive_selection.py

6.3 common.gui.enums.DriveSelectorMode Class Reference

Inheritance diagram for common.gui.enums.DriveSelectorMode:

Collaboration diagram for common.gui.enums.DriveSelectorMode:

Static Public Attributes

- int STANDARD = 0
- int WITH_KEYS = 1

6.3.1 Detailed Description

```
Enumeration for drive selector modes.

Attributes:
    STANDARD (int): Standard drive selection mode.
    WITH_KEYS (int): Drive selection mode with keys.
```

6.3.2 Member Data Documentation

6.3.2.1 STANDARD

```
int common.gui.enums.DriveSelectorMode.STANDARD = 0  [static]
```

6.3.2.2 WITH_KEYS

```
int common.gui.enums.DriveSelectorMode.WITH_KEYS = 1  [static]
```

The documentation for this class was generated from the following file:

- common/gui/enums.py

6.4 auxiliary_app.gui.key_generation_thread.KeyGenerationThread Class Reference

Inheritance diagram for auxiliary_app.gui.key_generation_thread.KeyGenerationThread:

Collaboration diagram for auxiliary_app.gui.key_generation_thread.KeyGenerationThread:

Public Member Functions

- __init__ (self, pin, drive_manager)
- run (self)

Public Attributes

- pin
- drive_manager

Static Public Attributes

- progress_update = pyqtSignal(str, int)
- status = pyqtSignal(RsaGenState, str)

6.4.1 Detailed Description

```
A QThread subclass responsible for generating RSA keys in a separate thread.
Signals:
    progress_update (str, int): Emitted to update the progress of the RSA key generation process.
    status (RsaGenState, str): Emitted to indicate the status of the RSA key generation process.

Attributes:
    pin (str): The PIN code used for RSA key generation.
    drive_manager (DriveManager): The drive manager instance used for managing drives during RSA key generatic

Methods:
    run(): Executes the RSA key generation process and emits progress and status updates.
```

6.4.2 Constructor & Destructor Documentation

6.4.2.1 __init__()

```
auxiliary_app.gui.key_generation_thread.KeyGenerationThread.__init__ (
              self,
              pin,
              drive_manager )
```

6.4.3 Member Function Documentation

6.4.3.1 run()

```
auxiliary_app.gui.key_generation_thread.KeyGenerationThread.run (
              self )
```

```
Executes the RSA key generation process in a separate thread.

This method updates the progress at various stages of the key generation
process and emits the final status upon completion or error.

Emits:
    progress_update (str, int): Updates the progress message and percentage.
    status (RsaGenState, str): Emits the final status of the key generation process.

Raises:
    Exception: If an error occurs during the key generation process.
```

6.4.4 Member Data Documentation

6.4.4.1 drive_manager

```
auxiliary_app.gui.key_generation_thread.KeyGenerationThread.drive_manager
```

6.4.4.2 pin

```
auxiliary_app.gui.key_generation_thread.KeyGenerationThread.pin
```

6.4.4.3 progress_update

```
auxiliary_app.gui.key_generation_thread.KeyGenerationThread.progress_update = pyqtSignal(str,
int)  [static]
```

6.4.4.4 status

```
auxiliary_app.gui.key_generation_thread.KeyGenerationThread.status = pyqtSignal(RsaGenState,
str)  [static]
```

The documentation for this class was generated from the following file:

- auxiliary_app/gui/key_generation_thread.py

6.5 auxiliary_app.gui.key_generator_window.KeyGeneratorWindow Class Reference

Inheritance diagram for auxiliary_app.gui.key_generator_window.KeyGeneratorWindow:

Collaboration diagram for auxiliary_app.gui.key_generator_window.KeyGeneratorWindow:

Public Member Functions

- __init__ (self)
- init_ui (self)
- open_pin_pad (self)
- start_key_generation (self, pin)
- update_progress (self, message, value)
- handle_status (self, status_code, message)
- close_application (self)

Public Attributes

- keygen_btn
- open_pin_pad
- quit_btn
- close_application
- drive_selection_widget
- progress_dialog
- keygen_thread
- update_progress
- handle_status

6.5.1 Detailed Description

```
A window for generating RSA keys with a graphical user interface.

Methods
-------
__init__():
    Initializes the KeyGeneratorWindow instance and sets up the UI.

init_ui():
    Sets up the user interface components, including buttons and layout.

open_pin_pad():
    Opens a PIN pad dialog for the user to enter a PIN before generating keys.

start_key_generation(pin):
    Starts the key generation process in a separate thread and shows a progress dialog.

update_progress(message, value):
    Updates the progress dialog with the current progress of the key generation.

handle_status(status_code, message):
    Handles the status updates from the key generation thread, showing appropriate messages.

close_application():
    Closes the application when the quit button is clicked.
```

6.5.2 Constructor & Destructor Documentation

6.5.2.1 __init__()

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.__init__ (
              self )
```

6.5.3 Member Function Documentation

6.5.3.1 close_application()

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.close_application (
              self )
```

```
Closes the application window.

This method logs an informational message indicating that the application
was closed by the user and then proceeds to close the application window.
```

6.5.3.2 handle_status()

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.handle_status (
              self,
              status_code,
              message )

Handles the status of the RSA key generation process.

Args:
status_code (RsaGenState): The current state of the RSA key generation process.
message (str): A message providing additional information about the status.

Actions:
- If the status_code is RsaGenState.ERRORED, closes the progress dialog and shows a critical error message.
- If the status_code is RsaGenState.FINISHED, closes the progress dialog and shows an informational success me
```

6.5.3.3 init_ui()

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.init_ui (
              self )

Initializes the user interface for the key generator window.
This method sets up the window title, geometry, and layout. It creates and configures
the buttons for generating RSA keys and quitting the application, as well as a drive
selection widget. The buttons are connected to their respective event handlers.
Widgets:
    - QPushButton: "Generate RSA Keys" button to initiate RSA key generation.
    - QPushButton: "Quit" button to close the application.
    - DriveSelectionWidget: Custom widget for drive selection.
Layout:
    - QVBoxLayout: Vertical layout to arrange the buttons and drive selection widget.
```

6.5.3.4 open_pin_pad()

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.open_pin_pad (
              self )

Opens a PIN pad dialog for the user to enter a PIN and starts the key generation process.
This method first checks if a drive is selected. If no drive is selected, it logs an
informational message and shows a warning message box to the user, then returns without
proceeding further. If a drive is selected, it opens a PIN pad dialog for the user to
enter their PIN. If the user successfully enters a PIN, it logs that the PIN was entered
and starts the key generation process using the entered PIN.

Returns:
    None
```

6.5.3.5 start_key_generation()

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.start_key_generation (
              self,
              pin )

Initiates the RSA key generation process and displays a progress dialog.

Args:
    pin (str): The PIN code required for key generation.
This method sets up a QProgressDialog to inform the user about the progress
of the key generation process. It then starts a KeyGenerationThread to
perform the actual key generation in the background. The progress of the
key generation is updated via the `progress_update` signal, and the status
is handled via the `status` signal.
```

6.5.3.6 update_progress()

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.update_progress (
              self,
              message,
              value )
```

```
Updates the progress dialog with a new message and progress value.

Args:
    message (str): The message to display on the progress dialog.
    value (int): The progress value to set on the progress dialog.
```

6.5.4 Member Data Documentation

6.5.4.1 close_application

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.close_application
```

6.5.4.2 drive_selection_widget

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.drive_selection_widget
```

6.5.4.3 handle_status

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.handle_status
```

6.5.4.4 keygen_btn

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.keygen_btn
```

6.5.4.5 keygen_thread

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.keygen_thread
```

6.5.4.6 open_pin_pad

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.open_pin_pad
```

6.5.4.7 progress_dialog

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.progress_dialog
```

6.5.4.8 quit_btn

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.quit_btn
```

6.5.4.9 update_progress

```
auxiliary_app.gui.key_generator_window.KeyGeneratorWindow.update_progress
```

The documentation for this class was generated from the following file:

- auxiliary_app/gui/key_generator_window.py

6.6 common.gui.pin_pad_dialog.PinPadDialog Class Reference

Inheritance diagram for common.gui.pin_pad_dialog.PinPadDialog:

Collaboration diagram for common.gui.pin_pad_dialog.PinPadDialog:

Public Member Functions

- __init__ (self)
- init_ui (self)
- add_number (self, number)
- clear_pin (self)
- backspace (self)
- get_pin (self)

Public Attributes

- pin
- pin_length
- pin_display
- accept
- clear_pin
- backspace

6.6.1 Detailed Description

```
A dialog window for entering a PIN code.

Attributes:
    pin (str): The current PIN entered by the user.
    pin_length (int): The maximum length of the PIN.

Methods:
    init_ui():
        Initializes the user interface of the dialog.
    add_number(number):
        Adds a number to the current PIN.
    clear_pin():
        Clears the current PIN.
    backspace():
        Removes the last digit from the current PIN.
    get_pin():
        Returns the current PIN.
```

6.6.2 Constructor & Destructor Documentation

6.6.2.1 __init__()

```
common.gui.pin_pad_dialog.PinPadDialog.__init__ (
              self )
```

Initializes a new instance of the PinPad dialog.

This constructor sets up the initial state of the PinPad dialog, including
initializing the pin to an empty string and setting the pin length to 4.
It also logs the creation of the PinPad instance and calls the method to
initialize the user interface.

Attributes:
    pin (str): The current pin entered by the user, initially an empty string.
    pin_length (int): The required length of the pin, set to 4.

6.6.3 Member Function Documentation

6.6.3.1 add_number()

```
common.gui.pin_pad_dialog.PinPadDialog.add_number (
              self,
              number )
```

Adds a number to the PIN if the current length of the PIN is less than the maximum allowed length.

Args:
    number (int): The number to be added to the PIN.

Side Effects:
    Updates the PIN display with the new PIN.
    Logs the number that was clicked.

6.6.3.2 backspace()

```
common.gui.pin_pad_dialog.PinPadDialog.backspace (
              self )
```

Removes the last character from the current PIN and updates the display.

If the PIN is not empty, this method removes the last character from the
`self.pin` attribute and updates the `self.pin_display` to reflect the
change. It also logs the current state of the PIN after the backspace
operation.

Returns:
    None

6.6.3.3 clear_pin()

```
common.gui.pin_pad_dialog.PinPadDialog.clear_pin (
              self )
```

Clears the current PIN.

This method resets the stored PIN to an empty string and updates the
display to show "PIN: ". It also logs an informational message indicating
that the PIN has been cleared.

6.6.3.4 get_pin()

```
common.gui.pin_pad_dialog.PinPadDialog.get_pin (
              self )
```

Retrieve the PIN.

Returns:
    str: The PIN code.

6.6.3.5 init_ui()

```
common.gui.pin_pad_dialog.PinPadDialog.init_ui (
              self )
```

Initializes the user interface for the PIN pad dialog.
This method sets up the window title, geometry, and layout for the PIN pad dialog.
It includes a display for the PIN, a grid layout for the number buttons (1-9),
and additional buttons for submitting, clearing, and backspacing the PIN input.
The stylesheet for the dialog is loaded from "common/gui/css/pin_pad.css".
Widgets:
    - QLabel: Displays the current PIN.
    - QPushButton: Number buttons (1-9) to input the PIN.
    - QPushButton: Submit button to accept the entered PIN.
    - QPushButton: Clear button to clear the entered PIN.
    - QPushButton: Backspace button to remove the last digit of the entered PIN.

6.6.4 Member Data Documentation

6.6.4.1 accept

```
common.gui.pin_pad_dialog.PinPadDialog.accept
```

6.6.4.2 backspace

```
common.gui.pin_pad_dialog.PinPadDialog.backspace
```

6.6.4.3 clear_pin

```
common.gui.pin_pad_dialog.PinPadDialog.clear_pin
```

6.6.4.4 pin

```
common.gui.pin_pad_dialog.PinPadDialog.pin
```

6.6.4.5 pin_display

```
common.gui.pin_pad_dialog.PinPadDialog.pin_display
```

6.6.4.6 pin_length

```
common.gui.pin_pad_dialog.PinPadDialog.pin_length
```

The documentation for this class was generated from the following file:

- common/gui/pin_pad_dialog.py

6.7 auxiliary_app.gui.enums.RsaGenState Class Reference

Inheritance diagram for auxiliary_app.gui.enums.RsaGenState:

6.8 main_app.gui.enums.SignState Class Reference

Inheritance diagram for main_app.gui.enums.SignState:

Collaboration diagram for main_app.gui.enums.SignState:

Static Public Attributes

- int FINISHED = 0
- int ERRORED = -1

6.8.1 Detailed Description

```
SignState is an enumeration representing the state of a signing process.

Attributes:
    FINISHED (int): Indicates that the signing process has completed successfully.
    ERRORED (int): Indicates that an error occurred during the signing process.
```

6.8.2 Member Data Documentation

6.8.2.1 ERRORED

```
int main_app.gui.enums.SignState.ERRORED = -1  [static]
```

6.8.2.2 FINISHED

```
int main_app.gui.enums.SignState.FINISHED = 0  [static]
```

The documentation for this class was generated from the following file:

- main_app/gui/enums.py

6.9 main_app.gui.sign_thread.SignThread Class Reference

Inheritance diagram for main_app.gui.sign_thread.SignThread:

Collaboration diagram for main_app.gui.sign_thread.SignThread:

Public Member Functions

- __init__ (self, pin, drive_manager, pdf_path)
- run (self)

Public Attributes

- pin
- drive_manager
- pdf_path
- rsa_key

Static Public Attributes

- progress_update = pyqtSignal(str, int)
- status = pyqtSignal(SignState, str)

6.9.1 Detailed Description

```
A QThread subclass to handle the process of signing a PDF file in a separate thread.
Signals:
    progress_update (str, int): Emitted to update the progress of the signing process.
    status (SignState, str): Emitted to update the status of the signing process.

Attributes:
    pin (str): The PIN code used for RSA key decryption.
    drive_manager (DriveManager): The drive manager instance to manage the drive operations.
    pdf_path (str): The file path of the PDF to be signed.

Methods:
    run(): Executes the signing process, emitting progress updates and status changes.
```

6.9.2 Constructor & Destructor Documentation

6.9.2.1 __init__()

```
main_app.gui.sign_thread.SignThread.__init__ (
              self,
              pin,
              drive_manager,
              pdf_path )
```

Initializes the SignThread class with the provided PIN, drive manager, and PDF path.

Args:
    pin (str): The personal identification number used for authentication.
    drive_manager (DriveManager): An instance of the DriveManager class to manage drive operations.
    pdf_path (str): The file path to the PDF document to be signed.

6.9.3 Member Function Documentation

6.9.3.1 run()

```
main_app.gui.sign_thread.SignThread.run (
              self )
```

Executes the signing process in a separate thread.

This method performs the following steps:
1. Emits a progress update indicating the start of RSA key decryption.
2. Decrypts the RSA key using the provided PIN and drive manager.
3. Emits a progress update indicating the start of PDF file signing.
4. Signs the PDF file using the decrypted RSA key.
5. Emits a progress update indicating the finalization of the process.
6. Emits a final progress update indicating completion.
7. Emits a status signal indicating the successful completion of the signing process.

If an exception occurs during any of these steps, it logs the exception and emits a status signal indicating a

Raises:
    Exception: If an error occurs during the signing process, it is caught and logged, and the status is updat

6.9.4 Member Data Documentation

6.9.4.1 drive_manager

```
main_app.gui.sign_thread.SignThread.drive_manager
```

6.9.4.2 pdf_path

```
main_app.gui.sign_thread.SignThread.pdf_path
```

6.9.4.3 pin

```
main_app.gui.sign_thread.SignThread.pin
```

6.9.4.4 progress_update

```
main_app.gui.sign_thread.SignThread.progress_update = pyqtSignal(str, int)  [static]
```

6.9.4.5 rsa_key

```
main_app.gui.sign_thread.SignThread.rsa_key
```

6.9.4.6 status

```
main_app.gui.sign_thread.SignThread.status = pyqtSignal(SignState, str)  [static]
```

The documentation for this class was generated from the following file:

- main_app/gui/sign_thread.py

6.10 main_app.gui.sign_and_verify.SignVerifyWindow Class Reference

Inheritance diagram for main_app.gui.sign_and_verify.SignVerifyWindow:

Collaboration diagram for main_app.gui.sign_and_verify.SignVerifyWindow:

Public Member Functions

- __init__ (self)
- init_ui (self)
- start_signing_file (self, pin, pdf_path)
- start_verifying_file (self, pub_key_path, pdf_path)
- update_progress (self, message, value)
- handle_status (self, status_code, message)
- verify_sign (self)
- sign_pdf (self)
- select_pdf_file (self)
- select_pub_key_file (self)
- close_application (self)

Public Attributes

- sign_button
- sign_pdf
- verify_button
- verify_sign
- quit_button
- close
- drive_selection_widget
- progress_dialog
- keygen_thread
- update_progress
- handle_status

6.10.1 Detailed Description

```
A window for signing and verifying PDF files.

Methods
-------
__init__():
    Initializes the SignVerifyWindow instance and sets up the UI.
init_ui():
    Sets up the user interface for the window, including buttons for signing, verifying, and quitting,
    as well as a drive selection widget.
start_signing_file(pin, pdf_path):
    Starts the process of signing a PDF file, showing a progress dialog and running the signing in a separate
start_verifying_file(pub_key_path, pdf_path):
    Starts the process of verifying a PDF file, showing a progress dialog and running the verification in a se
update_progress(message, value):
    Updates the progress dialog with the current progress message and value.
handle_status(status_code, message):
    Handles the status updates from the signing or verifying process, showing appropriate messages to the user
verify_sign():
    Initiates the process of verifying a PDF file by selecting the PDF and public key files and starting the v
sign_pdf():
    Initiates the process of signing a PDF file by opening a PIN dialog, selecting the PDF file, and starting
select_pdf_file():
    Opens a file dialog to select a PDF file for signing or verifying.
select_pub_key_file():
    Opens a file dialog to select a public key file for verifying a PDF.
close_application():
    Closes the application and logs the closure.
```

6.10.2 Constructor & Destructor Documentation

6.10.2.1 __init__()

```
main_app.gui.sign_and_verify.SignVerifyWindow.__init__ (
              self )
```

```
Initializes an instance of the Sign and Verify class.

This constructor calls the parent class's constructor, logs the creation
of the instance, and initializes the user interface.

Methods:
    init_ui: Initializes the user interface components.
```

6.10.3 Member Function Documentation

6.10.3.1 close_application()

```
main_app.gui.sign_and_verify.SignVerifyWindow.close_application (
              self )
```

```
Closes the application.

This method logs an informational message indicating that the application
was closed by the user and then proceeds to close the application window.
```

6.10.3.2 handle_status()

```
main_app.gui.sign_and_verify.SignVerifyWindow.handle_status (
              self,
              status_code,
              message )
```

Handles the status of signing or verifying a PDF file and displays appropriate message dialogs.

```
Args:
status_code (Enum): The status code indicating the result of the signing or verifying process.
                    Possible values are SignState.ERRORED, SignState.FINISHED, VerifyState.ERRORED, VerifyStat
message (str): The message to be displayed in the dialog.

Behavior:
- If the status code is SignState.ERRORED, closes the progress dialog and shows a critical error message box.
- If the status code is SignState.FINISHED, closes the progress dialog and shows an information message box.
- If the status code is VerifyState.ERRORED, closes the progress dialog and shows a critical error message box
- If the status code is VerifyState.FINISHED, closes the progress dialog and shows an information message box.
```

6.10.3.3 init_ui()

```
main_app.gui.sign_and_verify.SignVerifyWindow.init_ui (
              self )
```

```
Initializes the user interface for the PDF Signer & Verifier application.
This method sets up the main window's title, geometry, and layout. It includes
buttons for signing PDFs, verifying PDF signatures, and quitting the application.
Additionally, it adds a drive selection widget for selecting drives with keys.
UI Elements:
- Sign PDF Button: A button to sign a PDF document.
- Verify PDF Signature Button: A button to verify the signature of a PDF document.
- Quit Button: A button to close the application.
- Drive Selection Widget: A widget to select drives with keys.
The method also connects the buttons to their respective event handlers.
```

6.10.3.4 select_pdf_file()

```
main_app.gui.sign_and_verify.SignVerifyWindow.select_pdf_file (
              self )
```

```
Opens a file dialog for the user to select a PDF file.

Returns:
    str: The path to the selected PDF file, or None if no file was selected.
```

6.10.3.5 select_pub_key_file()

```
main_app.gui.sign_and_verify.SignVerifyWindow.select_pub_key_file (
              self )
```

```
Opens a file dialog for the user to select a public key file.
This method displays a file dialog that allows the user to choose a public key file.
If the user cancels the dialog or does not select a file, a warning message is shown
and the method returns None.

Returns:
    str or None: The path to the selected public key file, or None if no file was selected.
```

6.10.3.6 sign_pdf()

```
main_app.gui.sign_and_verify.SignVerifyWindow.sign_pdf (
              self )
```

Opens a dialog to enter a PIN, selects a PDF file, and starts the signing process.

This method performs the following steps:
1. Opens a PinPadDialog for the user to enter their PIN.
2. Logs the opening of the PinPadDialog.
3. If the user confirms the dialog, retrieves the entered PIN.
4. Logs the entered PIN.
5. Opens a file selection dialog for the user to select a PDF file.
6. If a PDF file is selected, starts the signing process with the entered PIN and selected PDF file.

Returns:
    None

6.10.3.7 start_signing_file()

```
main_app.gui.sign_and_verify.SignVerifyWindow.start_signing_file (
              self,
              pin,
              pdf_path )
```

Initiates the process of signing a PDF file.
This method sets up a progress dialog to inform the user about the signing process and starts a separate threa

Args:
    pin (str): The PIN code required for signing the PDF.
    pdf_path (str): The file path of the PDF to be signed.

Attributes:
    progress_dialog (QProgressDialog): A dialog to show the progress of the signing process.
    keygen_thread (SignThread): A thread that handles the signing process.

6.10.3.8 start_verifying_file()

```
main_app.gui.sign_and_verify.SignVerifyWindow.start_verifying_file (
              self,
              pub_key_path,
              pdf_path )
```

Starts the process of verifying a PDF file using a public key.
This method initializes a progress dialog to inform the user about the
verification process and starts a separate thread to handle the verification.

Args:
    pub_key_path (str): The file path to the public key used for verification.
    pdf_path (str): The file path to the PDF file to be verified.

6.10.3.9 update_progress()

```
main_app.gui.sign_and_verify.SignVerifyWindow.update_progress (
              self,
              message,
              value )
```

Updates the progress dialog with the given message and value.

```
Args:
    message (str): The message to display in the progress dialog.
    value (int): The progress value to set in the progress dialog.
```

6.10.3.10 verify_sign()

```
main_app.gui.sign_and_verify.SignVerifyWindow.verify_sign (
              self )
```

Verifies the digital signature of a selected PDF file using a selected public key file.
This method prompts the user to select a PDF file and a public key file. If both files are selected,
it initiates the verification process.

```
Returns:
    None
```

6.10.4 Member Data Documentation

6.10.4.1 close

```
main_app.gui.sign_and_verify.SignVerifyWindow.close
```

6.10.4.2 drive_selection_widget

```
main_app.gui.sign_and_verify.SignVerifyWindow.drive_selection_widget
```

6.10.4.3 handle_status

```
main_app.gui.sign_and_verify.SignVerifyWindow.handle_status
```

6.10.4.4 keygen_thread

```
main_app.gui.sign_and_verify.SignVerifyWindow.keygen_thread
```

6.10.4.5 progress_dialog

```
main_app.gui.sign_and_verify.SignVerifyWindow.progress_dialog
```

6.10.4.6 quit_button

```
main_app.gui.sign_and_verify.SignVerifyWindow.quit_button
```

6.10.4.7 sign_button

```
main_app.gui.sign_and_verify.SignVerifyWindow.sign_button
```

6.10.4.8 sign_pdf

```
main_app.gui.sign_and_verify.SignVerifyWindow.sign_pdf
```

6.10.4.9 update_progress

```
main_app.gui.sign_and_verify.SignVerifyWindow.update_progress
```

6.10.4.10 verify_button

```
main_app.gui.sign_and_verify.SignVerifyWindow.verify_button
```

6.10.4.11 verify_sign

```
main_app.gui.sign_and_verify.SignVerifyWindow.verify_sign
```

The documentation for this class was generated from the following file:

- main_app/gui/sign_and_verify.py

6.11 main_app.gui.enums.VerifyState Class Reference

Inheritance diagram for main_app.gui.enums.VerifyState:

Collaboration diagram for main_app.gui.enums.VerifyState:

Static Public Attributes

- int FINISHED = 0
- int ERRORED = -1

6.11.1 Detailed Description

```
Enum class representing the state of a verification process.

Attributes:
    FINISHED (int): Indicates that the verification process has finished successfully.
    ERRORED (int): Indicates that an error occurred during the verification process.
```

6.11.2 Member Data Documentation

6.11.2.1 ERRORED

```
int main_app.gui.enums.VerifyState.ERRORED = -1  [static]
```

6.11.2.2 FINISHED

```
int main_app.gui.enums.VerifyState.FINISHED = 0  [static]
```

The documentation for this class was generated from the following file:

- main_app/gui/enums.py

6.12 main_app.gui.verify_thread.VerifyThread Class Reference

Inheritance diagram for main_app.gui.verify_thread.VerifyThread:

Collaboration diagram for main_app.gui.verify_thread.VerifyThread:

Public Member Functions

- __init__ (self, pub_key_path, pdf_path)
- run (self)

Public Attributes

- pub_key_path
- pdf_path
- public_key

Static Public Attributes

- progress_update = pyqtSignal(str, int)
- status = pyqtSignal(VerifyState, str)

6.12.1 Detailed Description

```
A QThread subclass to handle the verification of a PDF file in a separate thread.
Signals:
    progress_update (str, int): Emitted to update the progress of the verification process.
    status (VerifyState, str): Emitted to update the status of the verification process.

Args:
    pub_key_path (str): The file path to the public key used for verification.
    pdf_path (str): The file path to the PDF file to be verified.

Methods:
    run(): Executes the verification process, emitting progress updates and status changes.
```

6.12.2 Constructor & Destructor Documentation

6.12.2.1 __init__()

```
main_app.gui.verify_thread.VerifyThread.__init__ (
              self,
              pub_key_path,
              pdf_path )
```

Initializes the VerifyThread instance with the provided public key path and PDF path.

Args:
    pub_key_path (str): The file path to the public key.
    pdf_path (str): The file path to the PDF document to be verified.

6.12.3 Member Function Documentation

6.12.3.1 run()

```
main_app.gui.verify_thread.VerifyThread.run (
              self )
```

Executes the verification process for a PDF file.

This method performs the following steps:
1. Emits a progress update indicating the start of reading the public key.
2. Reads the public key from the specified path.
3. Emits a progress update indicating the start of PDF file verification.
4. Verifies the PDF file using the provided public key.
5. Emits progress updates throughout the verification process.
6. Emits a final progress update upon completion.
7. Emits a status signal indicating the success or failure of the verification process.

Emits:
    progress_update (str, int): Signal to update the progress with a message and percentage.
    status (VerifyState, str): Signal to update the status of the verification process.

Raises:
    Exception: If an error occurs during the verification process, it is caught and logged, and the status is

6.12.4 Member Data Documentation

6.12.4.1 pdf_path

```
main_app.gui.verify_thread.VerifyThread.pdf_path
```

6.12.4.2 progress_update

```
main_app.gui.verify_thread.VerifyThread.progress_update = pyqtSignal(str, int)  [static]
```

6.12.4.3 pub_key_path

```
main_app.gui.verify_thread.VerifyThread.pub_key_path
```

6.12.4.4 public_key

```
main_app.gui.verify_thread.VerifyThread.public_key
```

6.12.4.5 status

```
main_app.gui.verify_thread.VerifyThread.status = pyqtSignal(VerifyState, str)  [static]
```

The documentation for this class was generated from the following file:

- main_app/gui/verify_thread.py

Chapter 7

File Documentation

7.1 auxiliary_app/__init__.py File Reference

Namespaces

- namespace auxiliary_app

7.2 auxiliary_app/gui/__init__.py File Reference

Namespaces

- namespace auxiliary_app
- namespace auxiliary_app.gui

7.3 auxiliary_app/utils/__init__.py File Reference

Namespaces

- namespace auxiliary_app
- namespace auxiliary_app.utils

7.4 common/__init__.py File Reference

Namespaces

- namespace common

7.5 common/drive_manager/__init__.py File Reference

Namespaces

- namespace common
- namespace common.drive_manager

7.6 common/gui/__init__.py File Reference

Namespaces

- namespace common
- namespace common.gui

7.7 common/logger/__init__.py File Reference

Namespaces

- namespace common
- namespace common.logger

7.8 common/utils/__init__.py File Reference

Namespaces

- namespace common
- namespace common.utils

7.9 main_app/__init__.py File Reference

Namespaces

- namespace main_app

7.10 main_app/gui/__init__.py File Reference

Namespaces

- namespace main_app
- namespace main_app.gui

7.11 main_app/utils/__init__.py File Reference

7.12 auxiliary_app/gui/enums.py File Reference

Classes

- class auxiliary_app.gui.enums.RsaGenState

Namespaces

- namespace auxiliary_app
- namespace auxiliary_app.gui
- namespace auxiliary_app.gui.enums

7.13 common/gui/enums.py File Reference

Classes

- class common.gui.enums.DriveSelectorMode

Namespaces

- namespace common
- namespace common.gui
- namespace common.gui.enums

7.14 main_app/gui/enums.py File Reference

Classes

- class main_app.gui.enums.SignState
- class main_app.gui.enums.VerifyState

Namespaces

- namespace main_app
- namespace main_app.gui
- namespace main_app.gui.enums

7.15 auxiliary_app/gui/key_generation_thread.py File Reference

Classes

- class auxiliary_app.gui.key_generation_thread.KeyGenerationThread

Namespaces

- namespace auxiliary_app
- namespace auxiliary_app.gui
- namespace auxiliary_app.gui.key_generation_thread

Variables

- auxiliary_app.gui.key_generation_thread.logger = logging.getLogger("global_logger")

7.16 auxiliary_app/gui/key_generator_window.py File Reference

Classes

- class auxiliary_app.gui.key_generator_window.KeyGeneratorWindow

Namespaces

- namespace auxiliary_app
- namespace auxiliary_app.gui
- namespace auxiliary_app.gui.key_generator_window

Variables

- auxiliary_app.gui.key_generator_window.logger = logging.getLogger("global_logger")

7.17 auxiliary_app/main.py File Reference

Namespaces

- namespace auxiliary_app
- namespace auxiliary_app.main

Variables

- auxiliary_app.main.logger = initialize(AUXILIARY_LOG_FILE)
- auxiliary_app.main.dev_manager = DriveManager()
- auxiliary_app.main.app = QApplication(sys.argv)
- auxiliary_app.main.window = KeyGeneratorWindow()

7.18 main_app/main.py File Reference

Namespaces

- namespace main_app
- namespace main_app.main

Variables

- main_app.main.logger = initialize(MAIN_LOG_FILE)
- main_app.main.dev_manager = DriveManager()
- main_app.main.app = QApplication(sys.argv)
- main_app.main.window = SignVerifyWindow()

7.19 auxiliary_app/utils/utils.py File Reference

Namespaces

- namespace auxiliary_app
- namespace auxiliary_app.utils
- namespace auxiliary_app.utils.utils

Functions

- auxiliary_app.utils.utils.generate_rsa_keys (pin, drive_manager, progress_signal=None)

Variables

- auxiliary_app.utils.utils.logger = logging.getLogger("global_logger")

7.20 common/utils/utils.py File Reference

Namespaces

- namespace common
- namespace common.utils
- namespace common.utils.utils

Functions

- common.utils.utils.load_stylesheet (widget, relative_path)

Variables

- common.utils.utils.logger = logging.getLogger("global_logger")

7.21 common/drive_manager/drive_manager.py File Reference

Classes

- class common.drive_manager.drive_manager.DriveManager

Namespaces

- namespace common
- namespace common.drive_manager
- namespace common.drive_manager.drive_manager

Variables

- common.drive_manager.drive_manager.logger = logging.getLogger("global_logger")

7.22 common/gui/drive_selection.py File Reference

Classes

- class common.gui.drive_selection.DriveSelectionWidget

Namespaces

- namespace common
- namespace common.gui
- namespace common.gui.drive_selection

Variables

- common.gui.drive_selection.logger = logging.getLogger("global_logger")
- int common.gui.drive_selection.DRIVES_REFRESH = 300

7.23 common/gui/pin_pad_dialog.py File Reference

Classes

- class common.gui.pin_pad_dialog.PinPadDialog

Namespaces

- namespace common
- namespace common.gui
- namespace common.gui.pin_pad_dialog

Variables

- common.gui.pin_pad_dialog.logger = logging.getLogger("global_logger")

7.24 common/logger/logger.py File Reference

Namespaces

- namespace common
- namespace common.logger
- namespace common.logger.logger

Functions

- common.logger.logger.compress_old_log (log_file)
- common.logger.logger.initialize (log_file)

Variables

- common.logger.logger.AUXILIARY_LOG_FILE = Path("auxiliary.log")
- common.logger.logger.MAIN_LOG_FILE = Path("main.log")
- common.logger.logger.ZIP_FILE = Path("logs.zip")

7.25 main_app/gui/sign_and_verify.py File Reference

Classes

- class main_app.gui.sign_and_verify.SignVerifyWindow

Namespaces

- namespace main_app
- namespace main_app.gui
- namespace main_app.gui.sign_and_verify

Variables

- main_app.gui.sign_and_verify.logger = logging.getLogger("global_logger")

7.26 main_app/gui/sign_thread.py File Reference

Classes

- class main_app.gui.sign_thread.SignThread

Namespaces

- namespace main_app
- namespace main_app.gui
- namespace main_app.gui.sign_thread

Variables

- main_app.gui.sign_thread.logger = logging.getLogger("global_logger")

7.27 main_app/gui/verify_thread.py File Reference

Classes

- class main_app.gui.verify_thread.VerifyThread

Namespaces

- namespace main_app
- namespace main_app.gui
- namespace main_app.gui.verify_thread

Variables

- main_app.gui.verify_thread.logger = logging.getLogger("global_logger")

7.28 main_app/utils/crypto_utils.py File Reference

Namespaces

- namespace main_app
- namespace main_app.utils
- namespace main_app.utils.crypto_utils

Functions

- RSA.RsaKey main_app.utils.crypto_utils.read_public_key (public_key_path)
- RSA.RsaKey main_app.utils.crypto_utils.decrypt_rsa_key (str pin, drive_manager, progress_signal=None)

Variables

- main_app.utils.crypto_utils.logger = logging.getLogger("global_logger")

7.29 main_app/utils/pdf_utils.py File Reference

Namespaces

- namespace main_app
- namespace main_app.utils
- namespace main_app.utils.pdf_utils

Functions

- main_app.utils.pdf_utils.sign_pdf (str pdf_path, RSA.RsaKey rsa_key, progress_signal=None)
- bool main_app.utils.pdf_utils.verify_pdf (str pdf_path, RSA.RsaKey public_key, progress_signal=None)
- main_app.utils.pdf_utils.check_pdf_exists (str pdf_path, progress_signal=None)
- main_app.utils.pdf_utils.initialize_signing_process (str pdf_path, progress_signal=None)
- main_app.utils.pdf_utils.read_pdf_file (str pdf_path)
- main_app.utils.pdf_utils.initialize_pdf_writer (str pdf_path)
- main_app.utils.pdf_utils.hash_pdf (bytes pdf_content, progress_signal=None)
- main_app.utils.pdf_utils.create_signature (RSA.RsaKey rsa_key, pdf_hash, progress_signal=None)
- main_app.utils.pdf_utils.add_signature_to_pdf (writer, reader, bytes signature, progress_signal=None)
- main_app.utils.pdf_utils.save_signed_pdf (str pdf_path, writer, progress_signal=None)
- main_app.utils.pdf_utils.read_pdf_metadata (str pdf_path, progress_signal=None)
- main_app.utils.pdf_utils.prepare_unsigned_pdf (reader, str pdf_path, progress_signal=None)
- main_app.utils.pdf_utils.verify_signature (RSA.RsaKey public_key, pdf_hash, bytes signature, str pdf_path, progress_signal=None)

Variables

- main_app.utils.pdf_utils.logger = logging.getLogger("global_logger")