Activity Report
Milestone 2
Group A - Topic 7

Riccardo Berni - Security
Francesco Paolo Liuzzi - Privacy
Davide Marchi - XAI

November 2024

1 Introduction

This report outlines the work done by Topic 7 about Milestone 2. The chosen working method was based on two main factors: the objectives of the milestone and the work of the other topics. In line with this approach, subgroups of requirements identified in the three Privacy, Security, and XAI reports produced for Milestone 1 were selected to optimize the satisfaction of these two factors.

This report is structured into three main sections: Privacy, Security, and XAI. Each section provides a detailed overview of the work of our topic.

2 Privacy

This section details the privacy-related interventions implemented for Milestone 2. We coordinated the work for this milestone with **Topic 2 (Databases)** and **Topic 4 (APIs)**.

The main part of the work focused on implementing access control, as required by the delivery, to avoid unauthorized access to any part of the system. Specifically, methods were implemented in the `api_auth.py` file to implement authentication to the API in a straightforward way. Specifically, we find the following methods:

- `retrieve_keys`: to retrieve securely a key from the database corresponding to a specified microservice (e.g. AI Agent) specified in the input.
- `get_verify_api_key`: it is an async method that allows us to easily implement the API access control wherever in the code. Verifies that the X-API-KEY specified during the API calls matches with the key(s) of the microservices allowed to access that API.
- `get_current_user`: to check if the currently authenticated user is allowed to access an API.

Below there is an example of how these methods work.

The following code shows an example of a dummy API endpoint accessible to everyone by executing a GET call.

```
@app.get("/smartfactory/dummy")
    def dummy_endpoint():
        return JSONResponse(content={"message": "Hello World!"},
            status_code=200)
```

Listing 1: Dummy API Endpoint

Let us suppose we want only the AI Agent to call the API. We modify the code as follows after having imported `get_verify_api_key` from **auth_api.py**:

```
@app.get("/smartfactory/dummy")
    def dummy_endpoint(api_key: str = Depends(get_verify_api_key(["
        ai-agent"])):
```

```
3         return JSONResponse(content={"message": "Hello World!"},
              status_code=200)
```

Listing 2: Dummy Authenticated API Endpoint

As we can see above, it is sufficient to specify a list of entities allowed to use the API. By defining a list of allowed key-value pairs for each entity that needs to use an API, we allowed everyone in the team to implement easily an authentication schema without knowing how to do that. In this way, we worked smartly without attending to the implementation of all the APIs to intervene with our work.

This intervention also required implementing a script to create on the fly a database table with all the dummy keys (i.e. just for testing purposes and to be replaced with the real ones) to be used with the authenticated API calls.

We also contributed to the implementation of register and login endpoints, producing a JWT token that securely contains user-related data to be used where user-verified access to the APIs is needed. For additional security, it has a time expiration. Now, the check can be implemented by everyone using the same logic as the previous code, so it does not require any prior knowledge of how to implement it.

In addition to this, every denied access to the APIs is logged, currently without saving any personal identifiable information with the reason of rejection.

2.1 Conclusion

The privacy-related work mainly focused on the implementation of access control mechanisms guaranteeing the satisfaction of the following GDPR principles/requirements defined in the privacy report of Milestone 1:

- API Authentication (GDPR Art. 5(1)(f))
- Role Based Access (GDPR Article 5(1)(b))
- Dynamic API Controls (GDPR Article 5(1)(b))
- Field Level Data Minimization (GDPR Art. 5(1)(c))
- Expiring Access (GDPR Art. 5(1)(e))
- Audit Trails (GDPR Art. 5(2))

3 Security

This section outlines the security requirements implemented for encrypting data in the database. We coordinated the work for this milestone with **Topic 2 (Databases)**. The primary objective is to enable encryption at rest for the Druid database. This is achieved by first generating an AES encryption key, then encrypting the data before storing it in Druid.

3.1 Key Creation and Storage Workflow

To securely store and manage the encryption keys, we use HashiCorp Vault. The workflow for creating and storing the key is as follows:

1. Start the Vault server.
2. Generate a random AES encryption key.
3. Store the key securely in Vault.
4. Stop the Vault server.

3.1.1 How to?

The Python script `Create_key.py` automates the following steps:

- Initializes and starts the Vault environment.
- Creates a random AES key and saves it in Vault.
- Stops the Vault server.

The encryption key is now securely stored in Vault. It can be retrieved using the function `retrieve_aes_key_from_vault(client)`, as shown in the following code snippet.

```
1 def retrieve_aes_key_from_vault(vault_client) -> bytes:
2     secret = vault_client.secrets.kv.v2.read_secret_version(path=
          vault_path)
3     aes_key_hex = secret['data']['data']['aes_key']
4     return bytes.fromhex(aes_key_hex)
```

3.2 Encryption Workflow

The **Cryptography** library was chosen for data encryption due to its simplicity and robust high-level functions, making encryption and decryption both secure and straightforward. The following steps are executed before submitting sensitive data to Druid:

1. Start the Vault server.
2. Retrieve the AES key previously stored in Vault.
3. Encrypt sensitive data using the AES key.
4. Stop the Vault server.

3.2.1 How to?

The Cryptography library provides a comprehensive set of cryptographic tools, including support for both symmetric and asymmetric encryption. A utility file named `Crypto_lib.py` has been developed to encapsulate functions for encryption, decryption, and key retrieval or storage.

The example below demonstrates the encryption function, which supports various data types such as integers, floats, and strings:

```python
def aes_encrypt(data: bytes, key: bytes = None) -> tuple:

    # Convert different types of data to bytes
    if isinstance(data, int):
        data_bytes = struct.pack('i', data)
    elif isinstance(data, float):
        data_bytes = struct.pack('d', data)
    elif isinstance(data, str):
        data_bytes = data.encode('utf-8')
    else:
        raise ValueError("Unsupported data type")

    # Create AES cipher object with CBC mode
    cipher = AES.new(key, AES.MODE_CBC)

    # Pad data to match AES block size (16 bytes)
    padded_data = pad(data_bytes, AES.block_size)

    # Encrypt data
    encrypted_data = cipher.encrypt(padded_data)

    return encrypted_data, cipher.iv, key
```

Before submitting data to Druid, the .csv file is converted into a dataframe. The sensitive columns are replaced with their encrypted counterparts using the functions in `Crypto_lib.py`.

3.3 Conclusion

To summarize the security-related work for Milestone 2, the workflow starts with the secure generation and storage of AES encryption keys using HashiCorp Vault. Sensitive data is encrypted with the Cryptography library prior to ingestion into Druid. By employing well-structured processes and robust cryptographic tools, this solution ensures compliance with data security standards while safeguarding sensitive information at rest.

The following security requirements outlined in the Milestone 1 report have been successfully implemented:

1. The system must encrypt personal data before storing it in the database, using AES as the encryption standard.

2. Passwords must be hashed upon user registration.

3. Provide Encryption at Rest, which protects data when stored in the database, safeguarding it from unauthorized access in the event of a storage system breach.

4. After access control, as for the login, decrypt the data when retrieving it from the database before using it, ensuring that the decryption key is securely managed and not exposed in the code or logs.

4 Explainable AI Functions

This section details the implementation of two Python programs, `XAI_rag.py` and `XAI_forecasting.py`, developed for use by different groups within the project:

- **Topic 5:** The RAG group will utilize `XAI_rag.py` to attribute responses generated by their systems to specific context sources.
- **Topic 3:** The Forecasting group will use `XAI_forecasting.py` to explain predictions made by time-series models.

These programs are designed to be callable and reusable, with each script including a `main()` function to demonstrate its functionality and facilitate testing.

4.1 `XAI_rag.py`: Response Attribution for RAG Systems

`XAI_rag.py` provides tools to attribute model-generated responses to specific segments of the provided context. This is essential for enhancing transparency in retrieval-augmented generation systems.

4.1.1 Functionality

The primary function, `attribute_response_to_context`, operates as follows:

1. Segments the AI-generated response into sentences using NLTK[1].
2. Segments the provided context into sentences.
3. Computes similarity scores between each response segment and all context segments using token set ratio[1] from RapidFuzz[2].
4. Attributes each response segment to the most similar context segment, provided the similarity exceeds a configurable threshold.

[1]String matching method that compares sets of tokens from two strings. It removes duplicates and ignores word order, calculating a similarity score between 0 and 100.

4.1.2 Code Highlights

Input Validation: Ensures correctness of input data.

```
def _validate_inputs(response, context, threshold, verbose):
    if not isinstance(response, str):
        raise ValueError("The 'response' parameter must be a string.")
    if not isinstance(context, list) or not all(isinstance(ctx, str) for ctx in context):
        raise ValueError("The 'context' parameter must be a list of strings.")
    if not (0 <= threshold <= 100):
        raise ValueError("The 'threshold' must be between 0 and 100.")
```

Listing 3: Input Validation in `XAI_rag.py`

Attribution Process: Matches response segments to the most similar context sentences.

```
if match:
    context_match, similarity_score, _ = match
    if context_match not in references:
        references.append(context_match)
        textExplanation += f'[{len(references)}] {context_match}\n'
    ref_num = references.index(context_match) + 1
    textResponse += response_segment + f'[{ref_num}] '
```

Listing 4: Similarity Matching in `XAI_rag.py`

Verbose Mode: Outputs detailed attribution information for debugging.

```
if verbose:
    print(f"Response Segment: {result['response_segment']}")
    if result['context']:
        print(f"Attributed Context: {result['context']}")
        print(f"Similarity Score: {result['similarity_score']:.2f}%")
```

Listing 5: Verbose Mode in `XAI_rag.py`

4.1.3 Usage

`XAI_rag.py` can be run directly or imported into another script. Example usage:

- Pass the response and context as inputs to the `attribute_response_to_context` function.
- Customize the similarity threshold for different requirements.

```
Text Response:
Artificial intelligence has revolutionized industries by improving
    processes in fields like transportation and healthcare.[1] For
    instance, machine learning allows systems to adapt and learn
    without being explicitly programmed.[2]

Text Explanation:
[1] Artificial intelligence (AI) has rapidly advanced in recent
    years.
[2] Machine learning, a subset of AI, focuses on developing
    algorithms that allow computers to learn from data without
    being explicitly programmed.
```

Listing 6: Output example of `XAI_rag.py`

4.2 `XAI_forecasting.py`: Explaining Forecasting Models

`XAI_forecasting.py` implements the LIME[3] framework to explain predictions made by time-series forecasting models, using the XAI360[4] wrapper. This allows the identification of features most influential to a model's predictions.

4.2.1 Functionality

The core function, `explain_model_with_lime`, performs the following:

1. Flattens the multidimensional time-series data into a format compatible with LIME.
2. Dynamically generates feature names for interpretability.
3. Perturbs the input data and generates explanations for the model's predictions.
4. (Optional) Visualizes the importance of features through horizontal bar plots.

4.2.2 Code Highlights

Flattening Data: Prepares time-series data for LIME.

```
training_data_flat = training_data.reshape(num_samples, seq_length
    * input_size)
feature_names = [f"Time_{t}_Feature_{f}" for t in range(seq_length)
     for f in range(input_size)]
```

Listing 7: Data Flattening in `XAI_forecasting.py`

Prediction Function: Adapts model predictions for LIME compatibility.

```
def predict_fn(data: np.ndarray) -> np.ndarray:
    inputs = data.reshape(batch_size, seq_length, input_size)
    inputs = torch.from_numpy(inputs).float().to(next(model.
        parameters()).device)
    with torch.no_grad():
        outputs = model(inputs)
    return outputs.cpu().numpy().flatten()
```

Listing 8: Prediction Function in `XAI_forecasting.py`

Visualization: Displays feature importance.

```
def plot_lime_explanation(exp, title="LIME Explanation"):
    features, importances = zip(*exp.as_list())
    plt.barh(features, importances, color="skyblue")
    plt.title(title)
    plt.xlabel("Importance")
    plt.ylabel("Feature")
    plt.show()
```

Listing 9: Feature Importance Visualization

4.2.3 Usage

`XAI_forecasting.py` is designed for ease of integration:

- Pass a trained PyTorch model and input data to the `explain_model_with_lime` function.
- Use verbose mode to generate plots for feature importance.

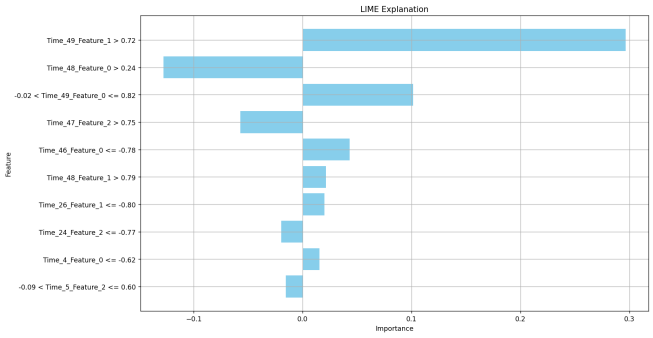

Figure 1: Output example of `XAI_forecasting.py`

4.3 Conclusion and Integration Plan

These tools are tailored to support the goals of Topics 3 and 5:

- `XAI_rag.py` provides attribution insights for retrieval-augmented generation workflows.
- `XAI_forecasting.py` offers interpretability for time-series forecasting models.

Next steps:

1. **Topic 5:** Integrate `XAI_rag.py` with the RAG pipeline to produce transparent and traceable outputs.
2. **Topic 3:** Leverage `XAI_forecasting.py` to explain model decisions, improving trust in forecasting predictions.

These programs are ready for use and further extension as needed.

References

[1] Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Toolkit (NLTK)*. Python library for natural language processing. Accessed: 2024-11-25. NLTK Project. 2001. URL: https://www.nltk.org/.

[2] Max Bachmann. *RapidFuzz Python Library*. A Python library for fuzzy string matching. Accessed: 2024-11-25. 2021. URL: https://github.com/maxbachmann/RapidFuzz.

[3] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 2016, pp. 1135–1144.

[4] IBM Research AI. *AI Explainability 360*. Accessed: 2024-11-08. 2023. URL: https://aix360.res.ibm.com/.