Attack Tree Visualiser

Overview

The Attack Tree Visualiser is a Python-based command-line application designed to support security professionals in analysing and visualising attack trees for risk modelling. This tool supports multiple file formats, including JSON, YAML, and XML, and allows users to assign either monetary impact or probabilities of success to threat nodes. It enables analysts to quantify risk and visualise attack pathways through colour-coded diagrams that highlight threat severity.

Key Features

- **Format versatility**: Accepts `.json`, `.yaml`, `.yml`, and `.xml` files. These widely supported formats allow seamless integration into existing workflows, from manually crafted trees to machine-generated data. They also offer flexibility when sharing attack models across teams and tools.
- **User interaction**: Prompts for updating leaf node values at runtime. This interactive prompt makes the tool adaptable to real-time analysis, allowing users to simulate different risk scenarios and assess how small changes in node-level threats impact the overall risk.
- **Mode flexibility**: Supports both monetary (e.g., £5000) and probability-based (e.g., 0.6) analysis. Analysts can switch between qualitative and quantitative analysis depending on the context or regulatory requirements.

- **Graph rendering**: Generates colour-coded visual diagrams using Matplotlib and Graphviz. The graphs intuitively communicate the structure and severity of attacks, using colour gradients to represent node impact.
- **Validation**: Automatically validates value formats and alerts users to errors. Incorrect formats or invalid values are caught early, which helps maintain data quality.
- **Modular architecture**: Cleanly separated modules for core logic, file I/O, graph computation, and plotting. This design promotes easier debugging, testing, and feature expansion.

Attack trees are used extensively in threat modelling and are considered foundational in identifying how various attacks can lead to a security breach.

Installation Instructions

1. Install Python

Ensure Python 3 is installed on your system.

2. Clone the repository

```
git clone
https://github.com/shraddha-gore/UoEO-attack-tree-visualiser.git
```

3. Navigate to the project directory

```
cd UoEO-attack-tree-visualiser
```

4. Install Graphviz

Graphviz is required for visualising the attack trees.

```
sudo apt-get install graphviz
```

5. Set up a Python virtual environment

```
python3 -m venv venv

source venv/bin/activate
```

6. Install dependencies

```
pip install -r requirements.txt
```

How to Use

Run the application using:

```
python3 main.py path/to/your/tree.yaml
```

Example:

```
python3 main.py data/probability/pre_probability.json
```

The user is prompted to:

1. Choose between monetary or probability interpretation.
2. Update values for each leaf node.
3. View the calculated total impact or probability.
4. See a visualised graph with coloured nodes.

Application Logic

The application's main control flow is handled in `core.py`, where user input, validation, and aggregation are orchestrated. This includes prompting the user to update leaf node values, validating their correctness based on the selected interpretation mode (monetary or probability), and calculating the final aggregated result.

The entry point of the application is `main.py`, which parses command-line arguments and handles early-stage error checking, such as detecting unsupported file formats. It also prompts the user to choose between value interpretation modes and delegates further execution to the core logic.

Graph construction is handled in `utils/graph_utils.py`, where a recursive function builds a directed attack tree graph using NetworkX. This file also includes functions to propagate values upward through the tree and compute the final aggregated result depending on the selected analysis mode.

The `utils/io_handler.py` module is responsible for loading and parsing input files. It supports multiple formats, including JSON, YAML, and XML, converting them into a standard Python dictionary structure that the rest of the system can use. A helper function recursively converts XML into the expected format, ensuring consistency across file types.

Visual rendering is managed in `visualisation/plot_tree.py`, which uses Matplotlib to generate colour-coded, labelled diagrams of the attack tree. Graph layout is optimised using Graphviz, which interfaces with the dot engine to produce hierarchical structures that are easy to interpret.

Each module is kept logically independent, with well-defined responsibilities. This modular architecture ensures the application is easy to test, debug, and extend in future versions.

Functional Demonstration

To help validate the tool, screenshots and videos have been generated to showcase various use cases. These include:

- The visualisation of a probability-based post-digitalisation attack tree.
- A monetary model simulation of a pre-digitalisation infrastructure.
- Invalid input errors (e.g., passing a string instead of a number).
- Unsupported file format detection.

These screenshots and videos serve as proof of both functional correctness and user experience. They demonstrate that errors are caught early and that value propagation and graph layout function as intended.

Testing

We use `pytest` for unit testing. All test files are located under the `tests/` directory, covering core logic, graph utility functions, and file parsing. Tests are designed to be self-contained and avoid reliance on external files. Instead, attack trees are constructed in memory to enhance reproducibility and independence.

To run all tests and generate a report, execute:

```
pytest -v tests/ > tests_report/pytest_report.txt
```

The test results will be saved to `test_reports/pytest_report.txt`.

Linter Integration

The project includes a `.pylintrc` configuration file to enforce code quality and maintain consistent style using `pylint`.

To run the linter and save the output:

```
pylint . > pylint_report.txt
```

The linter report will be found in `lint/pylint_report.txt`.

Unnecessary directories such as `venv/`, `data/`, and `__pycache__/`, etc., are excluded from linting via the `.pylintrc` configuration.

Error Handling

- Unsupported file types are caught early in `main.py`.
- Values that don't match the selected mode prompt clear error messages.
- Input errors like non-numeric entries or probabilities outside [0, 1] are validated using Python exceptions.
- Rootless graphs and empty trees are also caught to avoid rendering issues.
- The system ensures that the required keys (`name`, `value`, `children`) exist in every node.

Design Rationale

- **Modularity**: Separation of I/O, computation, and UI improves testability.
- **Validation-first**: Data validation occurs before any computation or rendering.
- **File format independence**: The system allows seamless switching between supported formats without code changes.
- **User-driven interaction**: Prompting the user for updates enhances flexibility and situational analysis.
- **Graph-based modelling**: Visualisations help in understanding attack propagation and defence weak points.

Python Packages Justification

- NetworkX is used to construct and manage the attack tree structure efficiently due to its extensive graph handling capabilities (NetworkX, n.d.).

- Matplotlib is employed to render visual attack trees with intuitive colour mapping and clear labelling (Matplotlib, n.d.).
- Graphviz enhances the graph layout by interfacing with the dot engine for clean hierarchical rendering (Graphviz, n.d.).
- YAML and JSON parsing are supported using PyYAML and Python's json module, both widely accepted for their stability and readability (Amazon Web Services, n.d.).
- Pylint is integrated to enforce consistent coding practices and detect potential issues during development (Hari, 2024).
- Pytest is used for unit testing because of its support for fixtures and clean syntax, making it ideal for modular test design (Docquin, 2023).

Limitations and Future Work

While the tool is functionally complete for basic risk modelling, several limitations exist:

- **No AND/OR logic**: The app currently assumes all child nodes contribute equally to the parent node. Future updates could incorporate logical operators for better attack path modelling.
- **No persistent output**: Visualisations are not saved automatically. Allowing export to image formats or integration with reporting tools like LaTeX or PDF would improve usability.
- **No severity tagging**: Nodes are only coloured based on value; no additional metadata (e.g., threat category or attacker profile) is visualised.
- **No GUI**: A graphical interface could expand accessibility for non-technical users.

- **No auto-import from CVE feeds**: Automatically building trees based on known vulnerabilities would dramatically improve real-world applications.

These enhancements would make the Attack Tree Visualiser suitable for use in professional penetration testing reports, automated SOC dashboards, and academic research projects.

Reference List

Amazon Web Services (n.d.) What's the Difference Between YAML and JSON?. Available at: https://aws.amazon.com/compare/the-difference-between-yaml-and-json/ (Accessed: 17 July 2025).

Docquin, A. (2023) *Mastering Unit Tests in Python with pytest: A Comprehensive Guide*. Available at: https://medium.com/@adocquin/mastering-unit-tests-in-python-with-pytest-a-comprehensive-guide-896c8c894304 (Accessed: 18 July 2025).

Graphviz (n.d.) *Graphviz*. Available at: https://graphviz.org/ (Accessed: 17 July 2025).

Hari, H. (2024) *Enhancing Python Code Quality with Pylint*. Available at: https://www.cloudthat.com/resources/blog/enhancing-python-code-quality-with-pylint (Accessed: 18 July 2025).

NetworkX (n.d.) *NetworkX*. Available at: https://networkx.org/(Accessed: 17 July 2025).

Matplotlib (n.d.) *Matplotlib*. Available at: https://matplotlib.org/stable/ (Accessed: 17 July 2025).