Project Title: Static Malware Analysis Automation Framework

Short Project Description:

The system analyzes malware binaries without execution, identifies indicators of compromise (IOCs), predicts behaviors, and generates actionable reports for security teams.

Component	Role
Malware Sample Collector	Collects executable samples (EXE, APK, ELF, etc)
Disassembler Module	Converts binary code into assembly
String and Signature Extractor	Pulls out strings, hashes, known patterns, etc
Behavioral Predictor	Predicts behavior based on static features
Reporting Engine	Prepares structured malware analysis reports

Component Details:

1. **Malware Sample Collector**:
   - Gathers samples from honeypots, malware feeds, submissions, etc.
2. **Disassembler Module**:
   - Uses disassemblers like **IDA Pro**, **Radare2**, **Ghidra, etc**.
   - Translates binaries into assembly code.
3. **String and Signature Extractor**:
   - Extracts:
     - Hardcoded URLs
     - IP addresses
     - API function calls
     - Magic constants (known malware indicators)
     - Etc
4. **Behavioral Predictor**:
   - ML model that predicts malware capabilities statically:
     - Keylogging, network beaconing, ransomware actions, etc
5. **Reporting Engine**:
   - Summarizes:
     - Static artifacts
     - Predicted behavior
     - Threat level

Overall System Flow:

- Input: Malware binary
- Output: Static malware analysis report
- Focus: **Preliminary triage without execution**.

Internal Functioning of Each Module:

1. Malware Sample Collector

- **Sources**:
  - Malware sharing platforms (e.g., VirusShare, MalwareBazaar, etc).
  - Honeypots deployed in public IP ranges.
- **Handling**:
  - Metadata tagging (hashes: SHA256, MD5, etc).
  - Storage in secure, isolated repositories, etc.
- **Automation Tip**:
  - Scheduled fetch and classification scripts (Python/Bash).

2. Disassembler Module

- **Disassemblers**:
  - **Ghidra** (open-source)
  - **IDA Pro** (commercial)
  - **Radare2** (scriptable)
  - **Etc**
- **Process**:
  - Identify architecture (x86, ARM, MIPS, etc).
  - Disassemble sections:
    - `.text` (code)
    - `.data` (global variables)
  - Output instruction flow (`MOV`, `CALL`, `JMP`, etc).

3. String and Signature Extractor

- **String Extraction**:
  - Extract readable ASCII and Unicode strings.
  - Analyze for:
    - IP addresses
    - URLs
    - Registry keys
    - Etc

- **Signature Matching**:
    - YARA rule matching:
        - Identify common malware families.
        - Match known cryptographic keys.

---

4. Behavioral Predictor

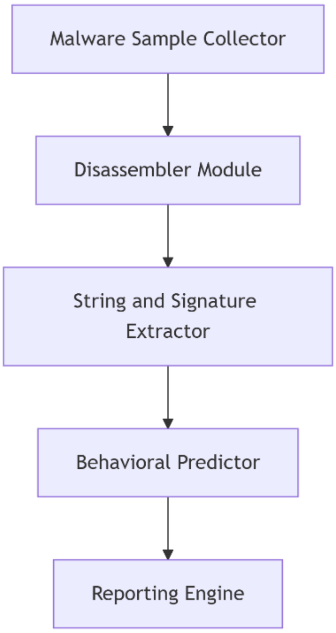
- **Models (e.g.)**:
    - Naive Bayes classifier for known API call patterns.
    - Neural Networks trained on n-grams of disassembly opcodes.
- **Examples**:
    - Heavy use of `CreateRemoteThread`, `VirtualAlloc` ➔ Indicates RAT.
    - Heavy use of `CryptEncrypt`, `WriteFile` ➔ Indicates ransomware.

---

5. Reporting Engine

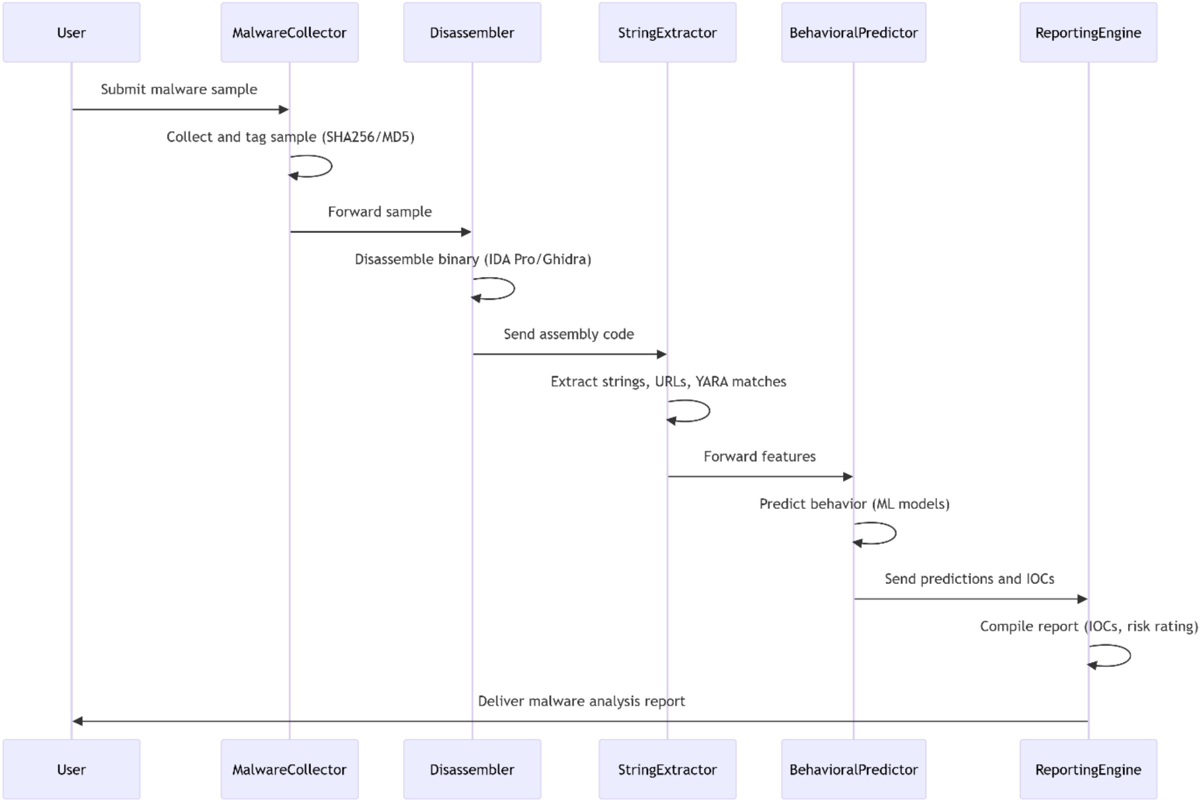
- **Sections**:
    - File metadata (hashes, size)
    - Static Indicators of Compromise (IOCs)
    - Predicted behavior
    - Risk rating (Low/Medium/High/Critical)
    - Etc

Component Diagram



- The **Malware Sample Collector** feeds binaries to the **Disassembler Module**.
- The **Disassembler Module** converts binaries into assembly code for the **String and Signature Extractor**.
- The **Behavioral Predictor** uses extracted features (strings, signatures, etc) to predict malware behavior.
- The **Reporting Engine** aggregates all data into a structured report for the user.

Sequence Diagram



- The **User** submits a malware sample, initiating the workflow.
- The **Malware Collector** tags and forwards the sample to the **Disassembler** for conversion into assembly.
- The **String Extractor** identifies static IOCs (URLs, IPs, YARA rules) and sends them to the **Behavioral Predictor**.
- The **Behavioral Predictor** applies ML models to predict capabilities (e.g., ransomware, keylogging).
- The **Reporting Engine** consolidates findings into a report detailing IOCs, predicted behavior, and risk ratings for the **User**.

Detailed Project Description: Static Malware Analysis Automation Framework

A framework for automating static malware analysis. The framework analyzes malware binaries without execution, identifies indicators of compromise (IOCs), predicts behaviors, and generates actionable reports for security teams.

---

1. System Overview

The framework performs static analysis of malware (EXE, APK, ELF, etc) to extract IOCs, disassemble code, and predict malicious behavior using machine learning. It focuses on **safe, execution-free triage** to support threat intelligence and incident response.

---

2. Component Design & Implementation

2.1 Malware Sample Collector

Functionality:

- Gathers malware samples from trusted sources and securely stores them.

Implementation Steps (e.g.):

1. **Sample Sources**:
    - **Malware Repositories**: Integrate APIs from VirusShare, MalwareBazaar, or Hybrid-Analysis, etc.
    - **Honeypots**: Deploy low-interaction honeypots (e.g., Cowrie, Dionaea, etc) to capture samples.
    - **User Submissions**: Build a secure upload portal (Flask/Django, etc).
    - **Etc**.
2. **Storage and Tagging**:
    - Store samples in isolated environments (Docker containers or VM snapshots).
    - Tag with metadata (SHA256, MD5, file type, submission date, etc).

3. **Automation**:
    - Schedule daily downloads using Python scripts (e.g., `requests` + `wget`).

Output:

- Catalog of malware samples with metadata.

Tools (e.g.):

- `Python`, `Docker`, `Flask`, `VirusShare API, etc`.

2.2 Disassembler Module

Functionality:

- Converts binaries into assembly code for analysis.

Implementation Steps (e.g.):

1. **Tool Integration**:
    - **Ghidra** (open-source): Use headless mode for batch disassembly.
    - **Radare2**: Script with `r2pipe` for automated analysis.

      ```
      import r2pipe
      r2 = r2pipe.open("malware.exe")
      disassembly = r2.cmd("pd 100")  # Disassemble first 100 instructions
      ```
    - **IDA Pro**: Use IDAPython for commercial-grade disassembly.
    - **Etc**
2. **Architecture Detection**:
    - Identify CPU architectures (x86, ARM, etc) via binary headers.
3. **Code Extraction**:
    - Extract `.text` (code) and `.data` (global variables) sections.

Output:

- Disassembled code (assembly/C pseudocode).

Tools (e.g.):

- `Ghidra`, `Radare2`, `IDA Pro`, `r2pipe, etc`.

2.3 String and Signature Extractor

Functionality:

- Extracts IOCs and matches malware signatures.

Implementation Steps (e.g.):

1. **String Extraction**:
    - Use `strings` command or `floss` (FireEye Labs Obfuscated String Solver) for obfuscated strings.
    - Regex for IOCs:
        - IPs: `\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`
        - URLs: `https?://[^\s]+`
        - Registry keys: `HKEY_\w+\\[^\s]+`
2. **Signature Matching**:
    - Write YARA rules for malware families:
    - Example

      ```
      rule Emotet_Loader {
        strings: $a = { 6A 40 68 00 30 00 00 6A 14 }  // Emotet shellcode pattern
        condition: $a
      }
      ```
    - Integrate VirusTotal YARA feed for known signatures.

Output:

- List of IOCs and YARA rule matches.

Tools (e.g.):

- `floss`, `YARA`, `regex`, `VirusTotal API, etc`.

2.4 Behavioral Predictor

Functionality:

- Predicts malware behavior using static features.

Implementation Steps (e.g.):

1. **Feature Extraction**:
   - **API Calls**: Extract Windows API usage (e.g., `CreateRemoteThread`, `VirtualAlloc`).
   - **Opcode N-grams**: Generate sequences of assembly instructions (e.g., `MOV-CALL-JMP`).
2. **Model Training**:
   - **Dataset**: Use EMBER (Endgame Malware BEnchmark for Research) or custom-labeled samples.
   - **Algorithms**:
     - **Random Forest**: For API call-based classification.
     - **LSTM/CNN**: For opcode sequence analysis.
     - **Etc**.
3. **Prediction**:
   - Deploy models with `scikit-learn` or `TensorFlow`:
   - Example

     ```
     from sklearn.ensemble import RandomForestClassifier
     model = RandomForestClassifier()
     model.fit(features, labels)
     prediction = model.predict([sample_features])  # e.g., "Ransomware"
     ```

Output:

- Predicted behaviors (e.g., keylogging, ransomware, C2 beaconing).

Tools (e.g.):

- `scikit-learn`, `TensorFlow`, `EMBER dataset, etc`.

2.5 Reporting Engine

Functionality:

- Generates structured reports with IOCs, predictions, and risk ratings.

Implementation Steps (e.g.):

1. **Report Structure**:
   - **Metadata**: File hashes, size, architecture, etc.
   - **IOCs**: URLs, IPs, registry keys, etc.
   - **Behavior Predictions**: Confidence scores and descriptions.
   - **Risk Rating**: Critical/High/Medium/Low based on impact.
2. **Formats**:
   - **HTML**: Use Jinja2 templates for interactive tables.
   - **PDF**: Convert HTML to PDF with `WeasyPrint`.
   - **STIX/TAXII**: For threat intelligence sharing.
   - **Etc**

Output:

- Professional report for incident response teams.

Tools (e.g.):

- `Jinja2`, `WeasyPrint`, `STIX/TAXII, etc`.

3. Technology Stack (e.g.)

- **Analysis Tools**: Ghidra, Radare2, YARA, etc.
- **ML**: scikit-learn, TensorFlow, EMBER dataset, etc.
- **Automation**: Python, Docker, Flask, etc.
- **Reporting**: Jinja2, WeasyPrint, STIX, etc.

4. Evaluation & Validation

1. **Accuracy Testing**:
   - Test on labeled datasets (e.g., 1,000 samples with known behavior).
   - Metrics: Precision, recall, F1-score, etc.
2. **False Positive Check**:
   - Validate predictions against dynamic analysis tools (e.g., Cuckoo Sandbox, etc).
3. **Performance**:
   - Measure disassembly speed (seconds per MB) and prediction latency.

5. Development Roadmap

1. **Phase 1**: Build Malware Collector and Disassembler.
2. **Phase 2**: Implement String/Signature Extractor and Behavioral Predictor.
3. **Phase 3**: Develop Reporting Engine and integrate STIX/TAXII.
4. **Phase 4 (optional)**: Validate on real-world malware samples and optimize.

6. Challenges & Mitigations (optional)

- **Obfuscation**: Use `floss` for deobfuscation and entropy analysis.
- **Scalability**: Parallelize disassembly with multiprocessing.
- **Model Accuracy**: Continuously retrain models with new samples.

7. Glossary

- **IOC**: Indicator of Compromise
- **YARA**: Pattern-matching tool for malware research

- **STIX/TAXII**: Standards for threat intelligence sharing
- **Opcode**: Low-level machine instruction
