National College of Ireland

Configuration Manual

MSc Research Project
MSc Cybersecurity

Manoj Santhoju
Student ID: 23394544

Msc in Cybersecurity
National College of Ireland

Supervisor: Khadija Hafeez

National College of Ireland
Project Submission Sheet
Msc in Cybersecurity

Student Name:	Manoj Santhoju
Student ID:	23394544
Programme:	MSc Cybersecurity
Year:	2025—2026
Module:	MSc Research Project
Supervisor:	Khadija Hafeez
Submission Due Date:	11 December 2025
Project Title:	Configuration Manual
Word Count:	2000
Page Count:	12

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Manoj Santhoju
Date:	11th December 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Contents

1 Introduction **1**
1.1 Purpose of this Document 1

2 System Requirements **1**
2.1 Hardware Requirements 1
2.2 Software Requirements 1

3 Installation Instructions **2**
3.1 Installation on Windows 2
3.2 Installation on Linux (Debian/Ubuntu) 3
3.3 Installation on macOS 3
3.4 Troubleshooting 4

4 Usage Guide **4**
4.1 Command Reference 4
4.2 Tutorial: Analyzing a Windows Dump 4
4.3 Tutorial: Linux Rootkit Discovery 5
4.4 Interpreting Output 5

5 Advanced Configuration **6**
5.1 Customizing Detection Rules 6
5.1.1 Adding a New Suspicious Process 6
5.1.2 Adding a Semantic Rule 7
5.2 Logging Configuration 7
5.3 Performance Tuning 7
5.3.1 Increasing Visual C++ Buffer 7
5.3.2 Symbol Caching 7

6 Troubleshooting and FAQ **8**
6.1 Common Installation Errors 8
6.1.1 Pip SSL Verification Failed 8
6.1.2 YARA Build Failure 8
6.2 Runtime Errors 8
6.2.1 "Symbol table not found" 8
6.2.2 "KDBG not found" (Windows) 8

7 Command Quick Reference **9**
7.1 Automated Setup Scripts 9
7.2 Automated Testing 9
7.3 Framework Information 9
7.4 OS Detection 9
7.5 Basic Analysis Commands 9
7.6 Output Format Options 10
7.7 Performance Experimentation 10
7.8 Cross-Platform Validation 10
7.9 Test Memory Dump Creation 11
7.10 Unit Testing 11

7.11 Complete Workflow Examples 11
7.11.1 Quick Incident Response 11
7.11.2 Detailed Investigation 11
7.11.3 Research Benchmarking 12
7.12 Command Summary Table 12

List of Figures

List of Tables

1 Hardware Requirements Table 2
2 CLI Command Options 5
3 Quick Reference for Common Commands 12

Configuration Manual

Manoj Santhoju
23394544

1 Introduction

This Configuration Manual serves as the definitive user guide for the **Unified Memory Forensics Framework**. It allows security analysts and system administrators to install, configure, and utilize the framework for detecting malware and anomalies in memory dumps from Windows, Linux, and macOS systems.

1.1 Purpose of this Document

The purpose of this document is to:

- Define the hardware and software prerequisites for running the framework.
- Provide step-by-step installation instructions for all supported operating systems.
- Document the Command Line Interface (CLI) arguments and usage patterns.
- Offer troubleshooting guidance for common installation issues.

2 System Requirements

Before installing the framework, ensure that your analysis workstation meets the following specifications. Note that analyzing large memory dumps (e.g., 32GB RAM) requires significant computational resources.

2.1 Hardware Requirements

2.2 Software Requirements

The framework is built on Python 3 and relies on several native libraries for symbol handling.

- **Operating System**:
  - **Windows**: Windows 10 (Version 2004+) or Windows 11.
  - **Linux**: Ubuntu 22.04 LTS, Debian 12, or equivalent (must support Python 3.8+).
  - **macOS**: macOS 12 (Monterey) or newer.

Component	Minimum Specification	Recommended Specification
Processor (CPU)	Dual Core 2.0 GHz	Quad Core i7/i9 or Apple Silicon (M1/M2/M3)
Memory (RAM)	8 GB	32 GB or higher (Should exceed the size of the target memory dump)
Storage	20 GB Free Space	500 GB NVMe SSD (Fast I/O is crucial for parsing large files)
Network	None (offline analysis)	Internet connection for fetching symbol files

Table 1: Hardware Requirements Table

- **Runtime Environment**: Python 3.8, 3.9, or 3.10. (Note: Volatility 3 has known issues with Python 3.11+ in some environments).
- **Dependencies**:
  - `git`: For cloning the repository.
  - `pip`: Python package manager.
  - `virtualenv`: Strongly recommended for dependency isolation.

3 Installation Instructions

This section details the installation process for Windows, Linux, and macOS. We recommend using a virtual environment to prevent conflicts with system packages.

3.1 Installation on Windows

Prerequisites: Ensure "Python 3" is installed from python.org and "Add Python to PATH" was selected during installation. You may also need "Microsoft Visual C++ Build Tools" if compiling certain dependencies like `yara-python`.

1. **Open PowerShell as Administrator**.
2. **Clone the Repository**:

```
git clone https://github.com/ManojSanthoju/Memory-Forensics.git
cd Memory-Forensics
```

3. **Create a Virtual Environment**:

```
python -m venv venv
.\venv\Scripts\Activate.ps1
```

(Note: If you get a script execution policy error, run `Set-ExecutionPolicy RemoteSigned`).

4. **Install Dependencies**:

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

5. **Verify Installation**:

```
unified-forensics --help
```

3.2 Installation on Linux (Debian/Ubuntu)

1. **Install System Dependencies**:

```
sudo apt update
sudo apt install python3-dev python3-pip python3-venv git build-essential
```

2. **Clone and Setup**:

```
git clone https://github.com/ManojSanthoju/Memory-Forensics.git
cd Memory-Forensics
python3 -m venv venv
source venv/bin/activate
```

3. **Install Project**:

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

3.3 Installation on macOS

Note: On Apple Silicon (M1/M2/M3), you may need Rosetta 2 if using x86-only dependencies, although mostly everything supports ARM64 now.

1. **Install Xcode Command Line Tools**:

```
xcode-select --install
```

2. **Setup Environment**:

```
git clone https://github.com/ManojSanthoju/Memory-Forensics.git
cd Memory-Forensics
python3 -m venv venv
source venv/bin/activate
```

3. **Install**:

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

3.4 Troubleshooting

- **Error: "Microsoft Visual C++ 14.0 is required"**
  Solution: Download the "Build Tools for Visual Studio" from Microsoft and install the "C++ desktop development" workload.
- **Error: "Symbol table not found"**
  Solution: Volatility 3 needs internet access to download symbol files (.json/.pdb) from the Microsoft/Linux symbol servers on the first run. Ensure your machine has internet access or manually place symbol files in the `volatility3/symbols` directory.
- **Permission Denied on Linux**
  Solution: Ensure you are running commands as a user with write access to the directory, or use `sudo` if writing to system folders (discouraged for Python packages).

4 Usage Guide

The Unified Forensics Framework provides a single command-line entry point: `unified-forensics`. This unified command handles the complexity of selecting the right underlying tool (Volatility vs Rekall) based on the target OS.

4.1 Command Reference

The basic syntax is:

```
unified-forensics analyze [OPTIONS] MEMORY_DUMP
```

4.2 Tutorial: Analyzing a Windows Dump

Suppose you have a memory dump from a Windows 11 workstation suspected of running a C2 beacon. The file is named `win11_infected.mem`.

1. **Basic Analysis (Auto-Detection)**: Run the tool without specifying the OS. It will scan the header for the "KDBG" value.

Option	Example	Description
MEMORY_DUMP	evidence.mem	The path to the raw memory file (Required).
--os-type	--os-type windows	Force specific OS profile. If omitted, the tool auto-detects. Values: windows, linux, macos.
--plugins	--plugins malware	Comma-separated list of post-processing plugins. Default: None.
--metrics	--metrics	Enable calculation of Precision/Recall (requires ground truth data).
--output	--output report.json	Specify custom output file path. Default: analysis_results/.
--format	--format table	Output format: json (default), table, summary.

Table 2: CLI Command Options

```
unified-forensics analyze win11_infected.mem
```

Output: The tool prints "Detected OS: Windows" and saves 'results.json'.

2. **Malware Detection**: Enable the malware plugin to scan for suspicious process names (like nc.exe or powershell.exe with encoded arguments).

```
unified-forensics analyze win11_infected.mem --plugins malware --format table
```

Output: A table will be displayed in the terminal highlighting any processes with a "High" threat score.

4.3 Tutorial: Linux Rootkit Discovery

For a Linux server dump (debian.mem), you might want to identify hidden processes.

```
unified-forensics analyze debian.mem --plugins malware,rootkit_check
```

The tool will: 1. Detect "Linux" from the vmlinuz signature. 2. Run Volatility's linux.pslist and linux.psscan. 3. The rootkit_check plugin will compare the two lists. Any PID found in psscan (memory pool) but not in pslist (active task structs) is flagged as a hidden rootkit process.

4.4 Interpreting Output

The default JSON output is structured as follows:

```
{
  "metadata": {
    "os_type": "windows",
```

```
    "analysis_time": "2024-12-08T12:00:00"
  },
  "processes": [
    {
      "pid": 4560,
      "name": "explorer.exe",
      "ppid": 1200,
      "start_time": "...",
      "cmdline": "C:\\Windows\\explorer.exe"
    }
  ],
  "malware_analysis": {
    "threat_level": "HIGH",
    "indicators": [
      {
        "type": "Suspicious Process",
        "value": "nc.exe"
      }
    ]
  }
}
```

- **metadata**: Confirms what the tool detected.
- **processes**: The normalized list of all running tasks.
- **malware_analysis**: Only present if `--plugins malware` was used. Contains specific flags.

5 Advanced Configuration

This section is intended for advanced users and developers who wish to customize the framework's behavior, add new detection rules, or tune performance for specific hardware environments.

5.1 Customizing Detection Rules

The core of the malware detection logic resides in the `plugins/malware_detector.py` file. The framework uses a list of Regular Expressions to identify suspicious process names and arguments.

5.1.1 Adding a New Suspicious Process

To flag a new process name (e.g., a specific ransomware variant), locate the `self.malware_patterns` list in the `__init__` method:

```
self.malware_patterns = [
    r'nc\.exe',
    r'powershell\.exe',
```

```
    # Add your new pattern below:
    r'my_ransomware\.exe',
]
```

5.1.2 Adding a Semantic Rule

For more complex logic (e.g., detecting a process running from a specific path), modify the `analyze` method. The `proc` dictionary contains standardized keys like `name`, `pid`, `ppid`, and `cmdline`.

```
# Example: Flag processes running from the Recyle Bin
if 'RECYCLER' in proc.get('path', ''):
    results['malware_indicators'].append({
        'type': 'Path Anomaly',
        'value': proc['name']
    })
```

5.2 Logging Configuration

By default, the framework logs standard information to the console. For debugging, you can enable verbose logging by modifying the `logging.basicConfig` level in `cli.py`.

- **INFO** (Default): Shows high-level progress (e.g., "Scanning...", "Detected OS").
- **DEBUG**: Shows detailed internal operations, including the exact volatile plugin commands being constructed and raw subprocess outputs.

```
# In cli.py
logging.basicConfig(
    level=logging.DEBUG,  # Change from INFO to DEBUG
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
```

5.3 Performance Tuning

Memory forensics is I/O intensive. When analyzing large memory dumps (32GB+), you may encounter timeouts or slow processing.

5.3.1 Increasing Visual C++ Buffer

If you see errors related to "buffer overflow" on Windows during the Volatility execution, you may need to increase the default subprocess buffer size in Python. This can be done by adjusting the `bufsize` parameter in the `subprocess.Popen` call within `core/framework.py`.

5.3.2 Symbol Caching

Ensure that the `volatility3/symbols` directory is on a fast SSD. Volatility heavily relies on JSON lookups within these symbol files. Moving the project to an NVMe drive can improve analysis speed by up to 40%.

6 Troubleshooting and FAQ

This section addresses common issues encountered during installation and usage, along with frequently asked questions.

6.1 Common Installation Errors

6.1.1 Pip SSL Verification Failed

Symptom: `pip install -r requirements.txt` fails with `[SSL: CERTIFICATE_VERIFY_FAILED]`. **Cause**: Your corporate firewall or antivirus is intercepting HTTPS traffic and ensuring its own self-signed certificate, which Python does not trust. **Solution**:

1. Temporarily trust the pypi host:

```
pip install --trusted-host pypi.org --trusted-host files.pythonhosted.org -r
```

2. Or, upgrade your `certifi` package: `pip install --upgrade certifi`.

6.1.2 YARA Build Failure

Symptom: `error: command 'gcc' failed` or `vcvarsall.bat not found`. **Cause**: The `yara-python` dependency requires a C compiler to build from source if a pre-compiled wheel is not available for your platform. **Solution**:

- **Windows**: Install "Visual C++ Build Tools" (see Section 3).
- **Linux**: Install `python3-dev` and `build-essential`.
- **macOS**: Run `xcode-select --install`.

6.2 Runtime Errors

6.2.1 "Symbol table not found"

Symptom: The analysis halts with an error indicating missing ISF (Intermediate Symbol Format) files. **Cause**: Volatility 3 could not identify the kernel version or download the required symbol file. **Solution**: 1. Check your internet connection (Volatility downloads symbols on first run). 2. Manually download the symbols (from the Volatility GitHub release) and place them in `volatility3/symbols/windows` (or linux/mac).

6.2.2 "KDBG not found" (Windows)

Symptom: The OS is detected as Windows, but the analysis returns 0 processes. **Cause**: The memory dump might be corrupted, or the KDBG structure is obfuscated (common in newer Win 10 builds). **Solution**: Try running the tool with the `--os-type windows` flag explicitly to bypass header checks, or try acquiring the memory dump again using a different tool (e.g., DumpIt instead of FTK Imager).

7 Command Quick Reference

This section provides a comprehensive reference of all available commands for installation, testing, analysis, and validation of the Unified Forensics Framework.

7.1 Automated Setup Scripts

The framework includes automated setup scripts for each supported platform:

```
# Windows
setup_windows.bat

# Linux/Ubuntu
chmod +x setup_linux.sh && ./setup_linux.sh

# macOS
chmod +x setup_macos.sh && ./setup_macos.sh
```

7.2 Automated Testing

Platform-specific test scripts validate framework functionality:

```
# Windows
test_complete_malware.bat

# Linux/Ubuntu
chmod +x test_complete_malware.sh && ./test_complete_malware.sh

# macOS
chmod +x test_complete_malware_macos.sh && ./test_complete_malware_macos.sh
```

7.3 Framework Information

```
# Display framework version and capabilities
python -m unified_forensics info
```

Expected output includes framework version, supported OS types, available plugins, and installed tools.

7.4 OS Detection

```
# Automatically detect OS from memory dump
python -m unified_forensics detect-os memory_dump.raw
```

7.5 Basic Analysis Commands

Listing 1: Analysis Commands

```
# Analyze with auto-detection
python -m unified_forensics analyze memory_dump.raw

# Specify OS type explicitly
python -m unified_forensics analyze memory_dump.raw --os-type windows

# Enable malware and network plugins
python -m unified_forensics analyze memory_dump.raw --plugins malware,netwo

# Enable metrics calculation
python -m unified_forensics analyze memory_dump.raw --plugins malware --me
```

7.6 Output Format Options

```
# JSON format (default)
python -m unified_forensics analyze memory_dump.raw --format json

# Human-readable table
python -m unified_forensics analyze memory_dump.raw --format table

# Concise summary
python -m unified_forensics analyze memory_dump.raw --format summary
```

7.7 Performance Experimentation

```
# Run performance experiments with default event rates
python -m unified_forensics experiment memory_dump.raw --os-type windows

# Custom event rates
python -m unified_forensics experiment memory_dump.raw --os-type windows \
  --rates 1 --rates 10 --rates 20 --rates 50 --rates 100

# Specify number of runs per rate
python -m unified_forensics experiment memory_dump.raw --os-type windows \
  --rates 1 --rates 10 --runs 10
```

7.8 Cross-Platform Validation

Listing 2: Validation Command

```
python -m unified_forensics validate \
    --windows-dump windows_test.raw \
    --linux-dump linux_test.raw \
    --macos-dump macos_test.raw
```

7.9 Test Memory Dump Creation

```
# Create Windows test dump
python create_malware_memory_dump.py \
  --filename test_windows.raw \
  --os-type windows \
  --size 50

# Create Linux test dump
python create_malware_memory_dump.py \
  --filename test_linux.raw \
  --os-type linux \
  --size 50

# Create macOS test dump
python create_malware_memory_dump.py \
  --filename test_macos.raw \
  --os-type macos \
  --size 50
```

7.10 Unit Testing

```
# Run all unit tests
python -m pytest tests/ -v

# Run specific test file
python -m pytest tests/test_framework.py -v

# Generate coverage report
python -m pytest tests/ --cov=unified_forensics --cov-report=html
```

7.11 Complete Workflow Examples

7.11.1 Quick Incident Response

```
# Fast triage during incident response
python -m unified_forensics analyze suspicious_dump.raw --format summary
```

7.11.2 Detailed Investigation

```
# Step 1: Detect OS
python -m unified_forensics detect-os evidence_dump.raw

# Step 2: Comprehensive analysis
python -m unified_forensics analyze evidence_dump.raw \
  --plugins malware,network \
  --metrics \
  --output investigation_results.json \
  --format table
```

```
# Step 3: Review results
cat investigation_results.json | python -m json.tool
```

7.11.3 Research Benchmarking

```
# Create test dumps
python create_malware_memory_dump.py --filename win.raw --os-type windows
python create_malware_memory_dump.py --filename lin.raw --os-type linux
python create_malware_memory_dump.py --filename mac.raw --os-type macos

# Run experiments
python -m unified_forensics experiment win.raw --os-type windows
python -m unified_forensics experiment lin.raw --os-type linux
python -m unified_forensics experiment mac.raw --os-type macos
```

7.12 Command Summary Table

Command	Purpose
`setup_windows.bat`	Install framework on Windows
`setup_linux.sh`	Install framework on Linux
`test_complete_malware.bat`	Run complete test suite
`python -m unified_forensics info`	Display framework information
`detect-os <dump>`	Detect OS type from memory dump
`analyze <dump>`	Analyze memory dump
`analyze <dump> --plugins malware`	Analyze with malware detection
`experiment <dump>`	Run performance experiments
`validate`	Cross-platform validation

Table 3: Quick Reference for Common Commands