Software Testing Plan (Version 2)

10/30/2025

Team Name: Cyber Recon

Sponsor: HighViz Security LLC

Team Mentor: Karthik Srivathsan Sekar

Team Members: Zachary Garza, Sean Weston, Jared Kagie, Christian Butler

Table of Contents

1 - Introduction 1

2 - Unit Testing 2

2.1 - Purpose 2

2.2 - Process, Tools, Metrics 2

2.2.1 - Coverage Targets 2

2.2.2 - Metrics Tracked 2

2.3 - Scope and Rationale 3

2. 4 - Unit Plans 3

NessusParser 3

SeverityAdjustmentClassifier 3

VulnerabilityProcessor 3

LearningSystem 4

2.5 - Execution and Reporting 4

2.6 - Summary 4

3 - Integration Testing 5

3.1 - Integration Test Cases 5

4 - Usability Testing 7

5 - Conclusion 9

1 - Introduction

Our application, Cyber Recon, was developed to address a very core problem our client, HighViz Security, is facing. When a new vulnerability is found, it's very time-intensive for our client to sift through countless hours of research when a program could research faster. Our solution enables users to research all unknown vulnerabilities and generate a severity rating that can closely match what a HighViz employee would say, providing a streamlined, automated, and efficient process that saves valuable time. The primary goal is to stay accurate to what a HighViz employee would research and evaluate all while reducing the amount of time a scan could take. Designed for our client specifically, this tool emphasizes accuracy, speed, and reliability.

Software testing is the process of evaluating and verifying that a software application performs as intended. Its main purpose is to detect defects, ensure functionality aligns with requirement documentation, and reliability validate functionalities under various conditions. Effective testing improves a user's trust with the program, minimizes the number of fixes that happen after deployment, and supports maintainability for a long period of time.

Our testing plan combines unit tests, integration tests, and usability testing to ensure comprehensive coverage of both functional and non-functional aspects of our solution. We will begin with unit testing the core functionality of our solution's requirements, followed by integration testing to confirm all data can flow freely through each of our modules. Usability testing will help make sure that each module makes the most sense for a user, allowing easy interactions with the program. Tools such as PyTest will be used to automate and validate results efficiently.

The emphasis on unit tests and integration tests stems from the nature of our application, which aims to be a modular system that focuses significantly less on features like a user interface. Since our client has more technical experience when it comes to programming, we prioritized integration tests and unit tests over others in order to ensure that our client can easily modify data if need be. Our testing regime aligns closely to what our client expects and desires from our product. Our solution ensures that accuracy, stability, and efficiency are validated in the areas most critical for world-use deployment.

The following sections provide a detailed explanation for each testing type, the specific procedures and tools applied, and the outcomes we expect from this testing plan. This plan not only ensures proper alignment towards requirements but ensures maintainability and stability throughout deployment.

2 - Unit Testing

2.1 - Purpose

Unit testing verifies the smallest testable parts of a codebase in isolation so defects surface quickly and are easy to localize. A unit can be a pure function, a method, or a small class with one responsibility. The goals are reliability, early defect detection, regression prevention, and safe refactoring, while also serving as executable documentation. Good unit tests are isolated from external systems, deterministic, focused on one behavior, readable, repeatable, and fast. They follow an arrange–act–assert flow and use test doubles where needed: stubs or fakes to supply data, and mocks or spies to verify essential interactions. Effective design relies on equivalence partitioning to cover representative input classes, boundary value analysis at edges and thresholds, property-minded checks for invariants, and explicit robustness tests for invalid types, nulls, missing fields, and malformed data. Coverage metrics guide scope rather than act as a goal: line coverage shows what ran, branch coverage exercises decision paths, and function or class coverage ensures public entry points are not missed. Unit tests form the foundation beneath integration and end-to-end tests, which are fewer and slower. Write tests before code when practical, add tests to capture existing behavior before refactors, and codify bugs as failing tests first so fixes cannot regress. Maintain quality by avoiding fragile assertions and over-mocking, keeping setup lean with fixtures, and reviewing tests as part of code review, so the suite remains fast, clear, and trustworthy.

2.2 - Process, Tools, Metrics

- Framework: pytest for discovery, fixtures, and parametrization
- Mocking: unittest.mock, pytest monkeypatch
- Coverage: pytest-cov
- Reporting: optional pytest-html
- Targets: line ≥ 85%, branch ≥ 75%, function ≥ 90% of public methods, class ≥ 80%
- Run: pytest tests/ --cov=src --cov-report=html
- CI: run on each commit, upload reports, fail if thresholds not met

2.2.1 - Coverage Targets

- Line coverage: at least 85 percent
- Branch coverage: at least 75 percent
- Function coverage: at least 90 percent for public methods
- Class coverage: at least 80 percent

2.2.2 - Metrics Tracked

- Coverage trends over time
- Suite runtime to catch performance regressions
- Flake rate for intermittent failures
- Diff coverage to ensure new or changed code includes tests

2.3 - Scope and Rationale

Detailed unit tests focus on high-risk areas. Full workflows and external APIs are covered by integration tests. Thin third-party wrappers and trivial utilities are not unit tested in depth.

Units with detailed tests:

- NessusParser
- SeverityAdjustmentClassifier
- VulnerabilityProcessor
- LearningSystem

2. 4 - Unit Plans

NessusParser

- Partitions: valid XML, malformed XML, empty scans, large scans, special characters
- Boundaries: 1 finding, 10,000 or more findings, severities 0 to 4, very long descriptions
- Selected inputs: one.nessus (1 row), empty.nessus (0 rows), corrupted.nessus (ValueError), large_scan.nessus (≥ 1,000 rows), specials.nessus (Unicode preserved)
- Erroneous: nonexistent path, binary masquerading as .nessus, missing required tags

SeverityAdjustmentClassifier

- Partitions: known patterns, novel patterns, text rich, text poor, high and low confidence
- Boundaries: single word, very long text, confidence near 0.3, 0.5, 0.7, all severity transitions
- Selected inputs: RCE phrasing, RAISE with high confidence; minimal “Vulnerability”, KEEP with low confidence; Unicode text processes cleanly
- Erroneous: empty string defaults safely, non string rejects if None, long text stays within limits

VulnerabilityProcessor

- Partitions: complete records, partial records, high risk vs low risk, known vs unknown CVEs
- Boundaries: CVSS at 0, 4, 7, 9, 10; single vs large batch
- Selected inputs: full record yields ai_risk_score, threat_intel, final_severity; partial record still produces final_severity; CVSS 0 maps to lowest tier, CVSS 10 to highest
- Erroneous: invalid CVSS clamped or flagged, malformed CVE unresolved with warning, API timeouts handled with fallback

LearningSystem

- Partitions: valid feedback, incomplete feedback, agreement vs corrective, sufficient vs insufficient data
- Boundaries: exactly minimum feedback for retraining, very large sets, drift windows 1 to 365 days
- Selected inputs: 10 entries, retrain True; 5 entries, retrain False with reason; conflicting feedback resolved by policy
- Erroneous: malformed IDs rejected, DB write failure rolled back, model save or load retried then reported

2.5 - Execution and Reporting

Local runs use the commands above. CI enforces coverage gates, tracks coverage trend and suite time, and flags flaky tests.

2.6 - Summary

This section defines what unit testing is and why it matters for Cyber Recon, lists the tools and metrics we use, and explains the scope of choices that prioritize critical units. It presents the units under test and, for each one, the equivalence of partitions, boundary values, and selected inputs we will use. It also includes erroneous input cases to show how robustness is validated. Together, the design, coverage targets, and CI gates provide a disciplined process that keeps parsing, classification, processing, and learning components correct, repeatable, and maintainable within the larger testing plan.

3 - Integration Testing

Integration testing involves combining different connected modules together to make sure data is sent between the modules as expected. This ensures that even when the individual modules function, when the modules are put together, they still function as intended. To determine what integration tests will need to be created we will find our major modules where important data is transferred from one to the other, such as our Data Processing module and AI/ML Engine module, and combine them into one test where test data is sent into one module and transferred through the other module(s). As for how we will implement integration testing, Pytest will be the main framework for creating and running the tests. Next, we will outline how we will implement integration testing.

We will implement an incremental approach to integration testing, which means that we will test the integration of modules in the order that they interact, to ensure that every module integrates properly from start to finish. Additionally, this allows us to determine where an integration test failed in our system, allowing us to quickly determine the source and cause of the failure and fix it quickly.

3.1 - Integration Test Cases

The first integration test case we will test is the integration between our nessus_parser and our processor, specifically making sure that the data generated by the nessus_parser is properly loaded into the processor module. To test this, we will simulate a blank environment where a nessus file is inputted but not processed. Next, we will use nessus_parser to parse the data into a csv file, then load these results into the processor module. We will then verify that the data now loaded into the processor module matches the csv as well as the nessus file, verifying that the data has been correctly parsed from the nessus file and loaded into the processor module.

Our next integration test case will be testing the integration between the processor module and the external database API modules. This will make sure that all the data that we need to analyze the provided nessus file is received from these sources, and that the data is correct and not missing any crucial information. This will be tested by creating a test dataset that would be from a nessus file, then calling the various APIs to bring in data related to our test dataset. If the data received matches the test dataset, and is correct and missing no information, then we will consider the test passed.

Next, we will test the integration between the processor module and the vulnerability_classifier module. Specifically, we will make sure that the AI scores from vulnerability_processor that are generated when provided with the data from the nessus scan make sense in context of the original data. To test this, we will take test data of both the parsed nessus scan and the API calls and feed them into the vulnerability_classifier. From there, when the vulnerability_classifier returns its data, we will make sure that it is properly recorded and stored in the processor module, and that the data makes sense in the context of the original data.

The final integration test will involve the integration between the processor module and the anomaly_detector module. In this case, the anomaly_detector module is called to find and return a list of anomalies, which are stored and recorded back into the processor module. The point of this test is to ensure that all anomalies found are properly reported to the processor. The test will be structured so that test data will be given to the anomaly_detector, which when returned and stored in the processor will be analyzed to ensure that it matches what anomalies are expected.

These tests aim to test not just whether each module functions as intended, but also that when these modules interact with each other, they continue to behave as expected. Additionally, by completing each of these integration tests in order, we can determine where an error occurs in the system, simply because we are following the program step by step as it moves between modules.

4 - Usability Testing

Usability testing is a critical evaluative tool that examines how effectively end users can interact with a software system to accomplish their intended goals. Unlike other testings, which verify that features work correctly, usability testing focuses on the quality of the user experience by assessing whether the interface is helpful, understandable, and complete for users to finish tasks efficiently without confusion or frustration. The primary goals include identifying inconsistencies in the user interface, validating that the system meets expectations, locating areas where users could make errors, and gathering informative feedback to improve the overall user experience.

For command-line interface (CLI) applications like Cyber Recon, usability testing focuses on how users interact with terminal commands and interpret text-based output. Without graphical elements, the system must communicate entirely through command structure, terminal messages, and output readability. Users must quickly understand what the system is doing, interpret the status, comprehend error messages, and extract information from the output. A well-designed CLI provides clear, scannable terminal output that presents critical information prominently. Poor CLI usability leads to cryptic messages or errors that users cannot discover, verbose output that do not expand on important messages, unclear progress indicators, or confusing error messages that prevent independent problem resolution.

Cyber Recon operates primarily through command-line interaction, with users executing commands like “python src/tools/process_nessus_scan.py --input-file scan.nessus --output-dir output --report-format xlsx” to process vulnerability scans. The system must provide clear terminal output during processing that communicates processing stages, operation duration, results found, and any issues requiring attention. The stakes for poor CLI readability are unusually high. If terminal output is unclear, analysts might miss critical information about KEV overrides, overlook anomaly detection warnings, misunderstand AI risk scoring rationale, or fail to notice when enrichment of APIs is unavailable. Given that Cyber Recon reduces analysis time from 8-12 hours to 60-90 seconds, terminal output must be immediately comprehensible. Our usability testing emphasizes how users discover and execute commands, whether terminal output is readable and scannable, if critical information stands out visually, whether progress indicators build confidence, if error messages enable independent troubleshooting, and whether users can quickly extract key metrics from CLI output.

The testing will occur over four weeks with internal team evaluation, cross-functional testing, and mentor/sponsor review. In the first week, team members who did not primarily develop the CLI interface will conduct systematic evaluation of command structure clarity and terminal output readability. Each evaluator will execute all primary commands documented in the README including the basic processing command, legacy mode comparison using --no-ai, verbose logging with --verbose, debug mode with --debug, threat intelligence updates, and intentionally triggered error scenarios such as missing files and invalid parameters. Team members will evaluate whether the help system provides sufficient guidance by attempting to complete tasks using only --help

output and README documentation. Each evaluator documents findings covering the specific command or output section affected, what made it unclear, severity, and recommended improvements.

During the second and third weeks, team members will rotate roles to provide fresh perspectives. Each team member will complete realistic vulnerability assessment scenarios using only the CLI, help documentation, and terminal feedback. Testing scenarios include processing initial scans, understanding AI decisions by reviewing composite risk scores and KEV overrides, testing model retraining workflows using --retrain-models --verbose, running severity adjustment model training with python src/tools/train_severity_adjustment_model.py, and comparing legacy mode --no-ai versus AI-enhanced mode. Team members will extract specific information from terminal output such as "How many Critical vulnerabilities were found?" and "Were any KEV overrides triggered?" to test whether output presents information clearly. Data collection focuses on time to discover commands, help consultations, time to extract information, command error rates, and qualitative feedback about readability.

In the fourth week, our team mentor will conduct comprehensive evaluation of CLI usability, executing documented workflows and providing expert feedback on command structure, terminal output organization, progress indicators, error messages, and overall user experience. If available, limited sponsor review with HighViz Security will validate that the CLI meets real-world professional expectations. Acceptance criteria include commands executable without constant documentation reference, terminal output scannable with critical information visually prominent, progress indicators informative and confidence-building, error messages enabling independent troubleshooting, and key metrics extractable in under 10 seconds.

All testing sessions will be recorded using terminal capture tools like asciinema. Team members will use structured observation forms to track CLI interaction events including commands attempted, help consultations, pauses while reading output, and successful information extraction. Quantitative analysis will measure time to discover and execute commands, percentage of output users read versus skim, time to locate specific information, and command error rates. Qualitative analysis will thematically code feedback into categories like "output too verbose" or "summary perfect for quick scanning". The team will create actionable improvements based on findings.

5 - Conclusion

In summary, this report outlined the testing strategy designed to ensure the quality and reliability of Cyber Recon, an AI-assisted vulnerability triage platform developed in collaboration with HighViz Security. After introducing the motivations behind the application and the fundamental goals of software testing, we detailed a structured plan incorporating tests like unit tests, integration tests, and usability tests. Each testing phase was made to address risk areas and to confirm that all critical components of the system perform as expected under real-world conditions.

By combining rigorous low-level testing with comprehensive integration and acceptance validation, this plan provides strong assurance that Cyber Recon will operate with minimal defects and maximum reliability. The balance of automated and manual testing supports efficient detection of edge cases while maintaining focus on usability and user satisfaction. This methodical approach aligns closely with our core objectives, ensuring that the final product is both robust and usable.

Additionally, the testing process for Cyber Recon reflects a deliberate effort to simulate the challenges faced by real security analysts who depend on accurate, timely, and interpretable data. For that reason, our tests are not limited to verifying that functions work in isolation but extend to validating the accuracy of machine learning outputs and the integrity of the data pipelines that feed them. Since Cyber Recon uses both structured and unstructured data sources, testing for consistent preprocessing, proper data labeling, and reliable cross-model communication was considered a top priority. These tests help prevent issues that might appear under realistic operating conditions, such as incomplete parsing or inconsistent prioritization scores.

Equally important is the emphasis on usability and workflow continuity. Because Cyber Recon is intended to enhance, not replace, human decision-making, user experience testing ensures that the interface and interactions support quick interpretation of results and easy navigation through vulnerability data. Feedback from early user testing cycles is incorporated into each iteration, allowing the platform to evolve alongside the needs of cybersecurity professionals who depend on it.

Currently, Cyber Recon has implemented a number of Unit Tests into the software application, as well as some mild integration testing. We plan to continue adding more integration tests while also implementing the previously said usability tests.

Taken together, these testing efforts create a layered quality assurance process that mirrors the complexity of the system itself. By aligning the intensity of testing with the nature of the application, this plan is well positioned to deliver a dependable and efficient product. The structured testing strategy, coupled with ongoing evaluation and refinement, establishes a strong foundation for long-term maintainability and scalability. With this testing plan in place, our development team can proceed confidently, knowing that each stage of verification contributes to delivering a reliable, high-performing, and accuracy-centered software solution that meets the demands of modern cybersecurity environments.