Final Capstone Report

Version 1

12/04/2025

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

Introduction

Cyber Recon is an AI-assisted vulnerability analysis platform developed for HighViz Security L.L.C to help their team of security analysts prioritize and sift through large amounts of vulnerability data quicker and more consistently. Modern organizations are facing an overwhelming number of findings from scanners, bug bounty programs, and external teams, in which human teams are expected to sort through all this information and decide what to fix first. Our project addresses this bottleneck directly, by building a repeatable, tweakable, machine learning process that efficiently classifies and prioritizes vulnerabilities with high accuracy, while staying flexible enough to evolve as the cyber landscape changes.

At the start of the year our version was very simple: take historical, labeled vulnerability data and use it to train a model that can support a faster, more accurate decision making than manual triage. Early in this project we had a straightforward solution to this, a single Random Forest classifier, trained on engineered features derived from vulnerability metadata and text fields. This baseline gives us the working proof-of-concept that we need, while also exposing real world challenges, such as class imbalance, noisy labels, and overfitting. Those challenges pushed our team to treat Cyber Recon less like a one-off product and more of a process that can be continuously tuned, retrained, and extended.

Over the course of the project, we constantly iterated our modeling approach. Prioritizing a severity classification system, we found that running a single Random Forest classification was not nearly enough. There were constant issues of overfitting, the model not accurately being able to represent minority classifications in the dataset, and adversely overfitting. Due to these issues, we moved from the single Random Forest to an ensemble of models, integrating techniques such as SMOTE (Synthetic Minority Over-Sampling Technique) to handle class imbalance and improve our performance on under-represented, but critical, classes. Each iteration followed a simple yet consistent workflow: clean and transform new data, retrain the ensemble, evaluate against held-out validation sets, and review results with our client. Throughout this entire process, we steadily improved classification quality, ultimately reaching around 91% accuracy on our cross-validation sets, while also reducing misclassifications in the high priority classes that matter most to analysts.

Equally important, our deliverable is not only a trained model artifact, but an end-to-end pipeline and development process that future teams can pick up on and evolve. Cybersecurity in nature is a dynamic field; new CVEs appear daily, exploitability changes, and client environments shift. For this reason, we focused on clean module data ingestion steps, easy feature engineering, and a training pipeline that can be run simply as new labeled data becomes available. Cyber Recon provides an automated and simple running of the legacy processes that HighViz already uses. Focusing on time efficiency, we have completely automated their current methods apart from the pipeline, while giving them the ability to retrain all within three intuitive commands. The goal is to give HighViz and its future developers a foundation, documented process, and not a static black box. They will have the ability to continue to refine the ensemble, plug in new models, change composite scoring systems, or adapt to new data sources without starting from scratch.

This final report will tell the story of how Cyber Recon was designed, built, and implemented as built. It begins with the problem context and requirements gathered from HighViz, then explains the system architecture and rationale behind our technical decisions. It will then go through the implementation details of the data and modeling pipeline, including the transition from the single Random Forest to an ensemble with SMOTE. Finally, it summarizes our evaluation results, discusses limitations and future work, and provides the practical information that a future software team will need to set up the environment, rerun the pipeline, and continue improving Cyber Recon beyond this Capstone project.

Process Overview

Over the course of the year, the project naturally fell into three overlapping phases: discovery and requirements, architecture and modeling, and integration and refinement. During the discovery phase, we spent a lot of time with our client learning how they currently class vulnerabilities, what slows them down, and what a "useful" model output looks like in their day-to-day work. That understanding shaped the way we designed the system and how we evaluated whether each change was an improvement, not just a better metric on paper. During the architecture and modeling phase, we built the initial data pipeline and baseline Random Forest model, then iterated on feature engineering, class balancing, and ensemble methods. In the integration and refinement phase, we focused on stabilizing the training pipeline, improving evaluation, and packaging the process so that a future team or HighViz engineers can retrain and extend the models.

Day-to-day work was organized using GitHub for version control and task management, supported by a standing weekly task report. All code lived in a shared GitHub repository, and we used a simple branching model where new features or experiments were implemented in short-lived branches, reviewed, and then merged into the main branch. The weekly task report listed both Capstone assignments (e.g., Requirements draft, Design updates, demo prep) and development tasks (e.g., "add SMOTE to training pipeline," "refactor feature engineering module," "evaluate ensemble v3"), along with an owner and a target week for completion. We used this report as the agenda for our weekly meetings, to check off completed items, carry over unfinished work, and add new tasks based on client feedback or instructor deadlines. GitHub Issues and project boards mirrored these items, so we had a record of work over the semester and a clear link between tasks and code changes.

We relied on a common Python-based toolchain for development. Team members used VS Code or similar editors, with a shared environment defined through dependency files so that everyone could run the same data pipeline and training scripts locally. Jupyter notebooks and small analysis scripts were used early on to explore data, prototype features, and compare different model configurations. Once an approach looked promising, we promoted it into the main training pipeline as a reproducible step. This helped us transition from ad-hoc experimentation to a repeatable process that our client and future teams can rerun when new data arrives.

Roles within the team were defined but flexible. One member acted as team lead and primary client liaison, responsible for scheduling meetings, organizing agendas, maintaining the weekly task report, and making sure client feedback flowed back into the backlog. Another member focused on data engineering, including cleaning raw vulnerability data, standardizing fields, and maintaining the feature generation steps. A third member took primary ownership of the modeling pipeline, including the baseline Random Forest, the later ensemble models, and the integration of SMOTE for class balancing. Our last team member took primary ownership of documentation, testing, and DevOps-adjacent tasks. Although shared, they took the "first owner" role to keep things moving.

Team procedures were captured in our informal standards: we agreed on consistent Python style, meaningful commit messages, and lightweight code review for nontrivial changes. For the modeling work specifically, we treated each major model configuration as a versioned experiment. New ensemble variants or SMOTE configurations were logged with their training parameters and evaluation metrics so that we could compare them and justify why we selected the final pipeline that achieved roughly 91 percent accuracy. This experimental discipline, combined with the weekly task reports and our GitHub workflow, is part of how we are delivering a process rather than just a single model file.

Throughout the year, we maintained a regular cadence of communication with HighViz through check-ins and milestone demos. Early demos focused on showing that we could ingest and normalize their data. Later demos highlighted the progression from a single Random Forest to an ensemble approach and the impact of SMOTE on minority classes. Feedback from these sessions directly influenced our priorities for upcoming sprints, such as emphasizing interpretability for analysts or reducing false positives in high-severity categories. By the end of the project, this process produced not only a working prototype, but also a clear, documented way for future teams to continue iterating on Cyber Recon in step with the evolving cybersecurity landscape.

Requirements

Our focus when it came to the decisions of the team's requirements was on the inefficiencies involved in current vulnerability assessment workflows. The current workflow of HighViz focuses on an initial vulnerability detection in client systems using an automated scanner called Nessus. However, manual review of Nessus data needs to be conducted for a more complete and correct severity rating through the process of long, tedious research that leads to slow response to threats and a risk of human error. This process is important for filtering out low priority findings but can allow for higher level risks to be overlooked or improperly identified. The solutions to these problems involved a way to automate vulnerability threat enrichment with real-time intelligence databases and prioritize accordingly based on current exploitability. The resulting requirements specify an artificial intelligence model with machine learning capabilities to learn and retrain while being able to run locally on HighViz Apple M2 and above hardware.

There are five main categories for the functional requirements: data ingestion, threat enrichment, AI risk assessment, output and intelligence management, and automation with human interaction inside the loop. Together, these requirements explain how our system functions in a complete process from Nessus scan to output file. Included with this are our non-functional requirements in which our goal is speed of the process, how much time our model takes to infer risk scores, memory usage, database query, boot timing, error resilience, and scalability.

To start with the data ingestion, the system must parse Nessus scan outputs in both CSV and JSON formats while accounting for schema variations across different Nessus versions. This first requirement is critical because the transfer of data to a readable format for the system allows for every other piece to work correctly. The parser validates each record for required fields such as plugin IDs, vulnerability titles, CVSS scores, and CVE identifiers, flagging any errors for review. Once parsed, the system normalizes severity levels by mapping numerical CVSS scores to standardized labels. These include HighViz's current interpretations from low severity and info to medium, high, and critical severity ratings.

The threat enrichment category focuses on augmenting raw scan data with external intelligence to provide deeper context for risk assessment. The system enriches CVE records with data from the National Vulnerability Database, including vulnerability summaries and updated CVSS metrics. It integrates Exploit Prediction Scoring System scores to find the probability of a vulnerability exploit being found in the near future. Additionally, the system uses CISA's Known Exploited Vulnerabilities database to assign higher concerns, flagging any CVE listed in the KEV as a critical, actively exploited threat. These enrichments transform the scan output into data that reflects real-world threats.

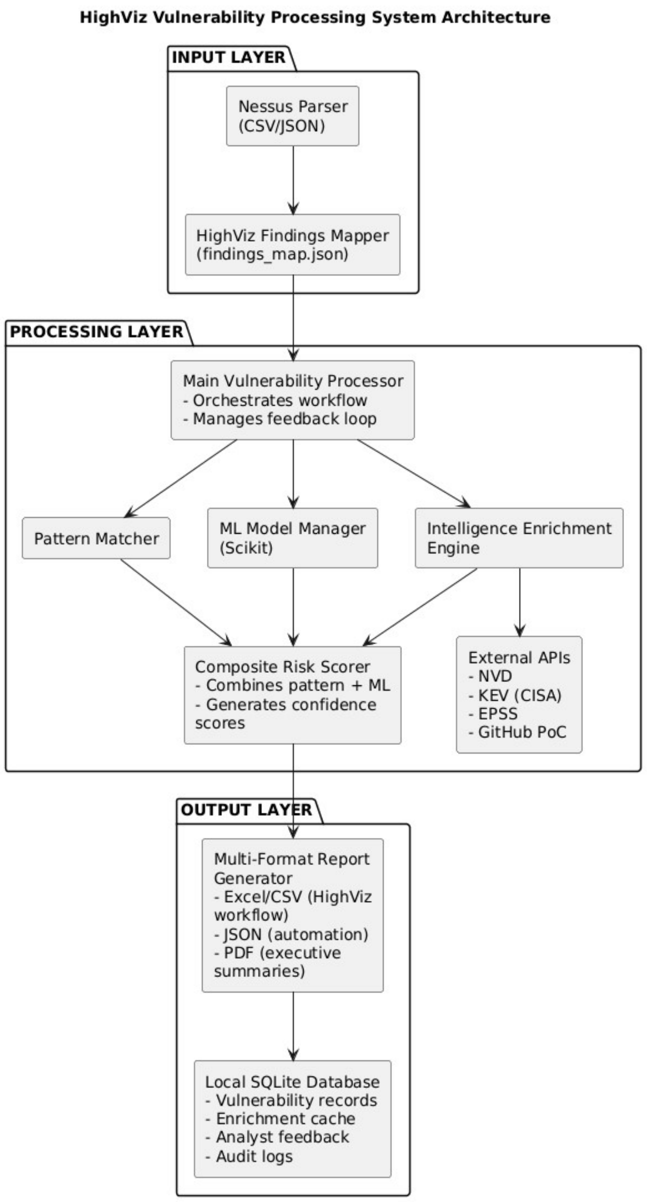
The AI risk assessment category is what acts on the previous enrichment category for recategorization of severities. The system employs NLP-based risk inference using a fine-tuned BERT model that analyzes vulnerability descriptions to identify key phrases indicating exploitability. These phrases can include "exploits exist", "remote code execution", or others that are read in a similar language. This textual analysis captures nuanced risk factors that numerical scores alone cannot convey. Complementing this, a structured machine learning model in the use of Random Forest helps evaluate quantitative features including CVSS scores, EPSS values, KEV flags, and CWE tags to produce numerical risk assessments. The system then generates composite risk scores by combining pattern recognition from HighViz's existing findings database with multi-model outputs, using an ensemble method to produce an explainable risk rating.

The output and intelligence management category ensures that processed data is presented in a simplistic manner for HighViz's clients. The system generates risk-based reports in Excel and CSV formats, featuring executive summaries alongside detailed vulnerabilities complete with affected hosts, risk levels, CVE summaries, EPSS scores, KEV flags, and AI justifications. These reports are designed for analysts to be able to send out their findings to clients with minimal changes being made. To support ongoing operations, the system maintains a local threat intelligence database using CSV files that store vulnerability data, threat feeds, AI model outputs, and analyst feedback. This database implements backup functionality and comprehensive audit logging to ensure traceability and recovery options.

The automation with human interaction category closes the loop between AI outputs and expert validation. The system provides an analyst feedback interface through CLI with GUI capabilities, allowing HighViz to review, correct, and annotate AI feedback. These annotations are captured with CVE plugin IDs, feeding directly into model retraining pipelines to continuously improve accuracy based on real-world validations. The system also implements a rule enforcement mechanism that allows manual overrides of AI outputs based on HighViz's specific company policies. When analysts update the findings map with their own scoring based on research, the system creates backups before applying changes. This ensures that domain expertise can correct generated assessments without risk of data loss. Finally, end-to-end automation is achieved through a master CLI pipeline that orchestrates the entire workflow, from file parsing through enrichment, risk inference, composite scoring, and report generation.

The non-functional requirements establish concrete performance benchmarks essential for operational viability. The processing speed of the system can process many entries within less than five minutes on a MacBook Pro M2 with 16GB of RAM. The model inference time describes how Cyber Recon can compute the risk scores fast, about 100 milliseconds per vulnerability. Memory usage aims to be below 2GB for low amounts of overhead on the system. Database query is focused on having under 50 milliseconds of latency for CVE lookups. Boot time has a 15 second limit for startup, minimizing the overhead through lazy loading and sequencing. Error resilience is handled through a comprehensive exception handling and logging of corrupted or unavailable records for clear notifications and understandable actions for recovery. Lastly, scalability emphasizes the maintainability of the system through file structure and modularity, as well as a documented usage guide.

Architecture & Implementation


HighViz Vulnerability Processing System Architecture
INPUT LAYER
Nessus Parser (CSV/JSON)
HighViz Findings Mapper (findings_map.json)
PROCESSING LAYER
Main Vulnerability Processor - Orchestrates workflow - Manages feedback loop
Pattern Matcher
ML Model Manager (Scikit)
Intelligence Enrichment Engine
Composite Risk Scorer - Combines pattern + ML - Generates confidence scores
External APIs - NVD - KEV (CISA) - EPSS - GitHub PoC
OUTPUT LAYER
Multi-Format Report Generator - Excel/CSV (HighViz workflow) - JSON (automation) - PDF (executive summaries)
Local SQLite Database - Vulnerability records - Enrichment cache - Analyst feedback - Audit logs

The vulnerability analysis system is designed as a three-layer pipeline architecture with modular components that enrich the Nessus scan data into an actionable and prioritized report. The system employes a machine learning approach to classify vulnerabilities, enriched with real-time threat intelligence from multiple external sources. As shown in the diagram, the three main layers are the input layer in charge of the system Nessus scans, and HighViz mapping file. After that is the processing layer which is the main orchestrator for machine learning and threat intelligence. Lastly, there is the output layer in which the information is put inside a multi-format report where the user can choose a file type.

Input Layer Components

1. Nessus Parser

Responsibilities:

- Ingests Nessus scan outputs in both CSV and JSON formats
- Normalizes schema differences across Nessus versions
- Extracts critical fields: PluginID, CVE identifiers, CVSS scores, vulnerability descriptions, affected hosts, ports

Key Features:

- Schema Mapping Engine: Configurable mapping rules handle header variations (e.g., "plugin_id" vs. "pluginID")
- Validation Pipeline: Checks each record for required fields, logs missing data
- Error Resilience: Skips corrupted records while continuing to process valid entries
- Format Detection: Automatically identifies input format and applies appropriate parser

Typical Use Case: When a HighViz engineer uploads a Nessus .csv file with 5,000 vulnerability findings, the parser:

1. Validates file integrity
2. Maps columns using the configuration file
3. Extracts PluginID, CVE, CVSS, description fields
4. Logs 3 records with missing CVE data
5. Outputs 4,997 normalized VulnerabilityRecord objects to the processing layer

2. HighViz Findings Mapper

Responsibilities:

- Integrates HighViz's proprietary knowledge base (findings_map.json)
- Maintains company-specific severity adjustments and vulnerability patterns
- Provides the "ground truth" for pattern matching

Key Features:

- Automatic Backup System: Creates timestamped backups before any modifications
- Version Control: Tracks changes to the findings map over time
- Fast Lookup: Indexed by PluginID retrieval
- Feedback Integration: Accepts analyst corrections and updates the map

Typical Use Case: When processing vulnerabilities:

1. Looks up the plugin in findings_map.json
2. Finds a historical match where HighViz previously elevated this from Medium to High
3. Returns the adjusted severity and justification
4. This becomes the primary classification, skipping ML inference

Processing Layer Components

1. Main Vulnerability Processor

Responsibilities:

- Orchestrator: Coordinates the entire vulnerability analysis workflow
- Decision Router: Determines which processing path each vulnerability takes (pattern match vs. ML inference)
- Feedback Manager: Handles analyst corrections and triggers model retraining

Key Features:

- Pipeline State Management: Tracks each vulnerability through the processing stages
- Parallel Processing: Uses multiprocessing to handle large datasets efficiently
- Error Recovery: Implements checkpointing to resume processing after failures
- Metrics Collection: Logs processing time, classification confidence, and resource usage

Typical Use Case: Processing a batch of 100 vulnerabilities:

1. 40 vulnerabilities match known patterns with high confidence - direct classification
2. 60 vulnerabilities are unknown - enriched with threat intel, then passed to ML model
3. All 100 receive final risk scores with confidence levels and explanations

2. Pattern Matcher

Responsibilities:

- Primary Classification Method: Matches Nessus findings against HighViz's historical knowledge base
- Signature Recognition: Identifies vulnerabilities by PluginID, description patterns, and characteristic features

Key Features:

- Multi-Field Matching: Combines PluginID, vulnerability description, affected software, and version numbers
- Fuzzy Matching: Uses Levenshtein distance for description similarity when PluginID doesn't exist in findings_map
- Confidence Scoring: Returns confidence levels (0.0 - 1.0) based on match quality
- Deduplication Logic: Prevents data leakage by ensuring training data doesn't contain duplicates

Typical Use Case: A vulnerability with a similar description to other vulnerabilities and unknown PluginID:

1. No exact PluginID match found
2. Fuzzy matching finds similar description with 0.82 similarity
3. Returns historical classification "Critical" with 0.82 confidence
4. Pattern match accepted, ML inference skipped

3. ML Model Manager

Responsibilities:

- Model Lifecycle Management: Loads, caches, and serves trained machine learning models
- Inference Execution: Runs predictions on enriched vulnerability data
- Model Retraining: Incorporates analyst feedback to improve accuracy over time

Key Features:

- Model Registry: Maintains multiple model versions with A/B testing capability
- Feature Engineering Pipeline: Transforms raw vulnerability data into ML-ready features
- Explainability: Uses SHAP values to explain which features drove each prediction
- Anomaly Detection: Flags unusual vulnerabilities that don't fit training distribution

Feature Set: The model is trained on the following features:

- CVSS Base Score (numerical)
- Tenable Plugin Severity (categorical)
- KEV Flag (binary: 0/1)
- EPSS Score (numerical, 0.0-1.0)
- CWE Category (categorical, one-hot encoded)
- GitHub PoC Availability (binary: 0/1)
- Vulnerability Age (days since CVE publication)
- Affected Software Category (categorical: OS, Web Server, Database, etc.)

Training Process: The initial model was trained on historical HighViz vulnerability assessments with the following pipeline:

1. Data Cleaning: Removed duplicates by PluginID + description to prevent data leakage
2. Feature Engineering: Applied standardization to numerical features, one-hot encoding to categorical
3. Train/Test Split: 80/20 split stratified by severity class
4. Model Selection: Random Forest chosen over XGBoost for interpretability and performance balance
5. Hyperparameter Tuning: Grid search optimized n_estimators=200, max_depth=15, min_samples_split=10
6. Validation: Achieved 87% accuracy on held-out test set

Addressing Overfitting Challenge: Early models showed 98.6% accuracy due to data leakage from duplicate PluginIDs. The solution:

- Deduplicated training data by (PluginID + description)
- Added vulnerability categories to improve generalization
- Implemented cross-validation with strict deduplication
- Final model accuracy: 87% (realistic and generalizable)

Typical Use Case: For an unknown vulnerability with CVSS 7.5, no KEV flag, EPSS 0.12, and CWE-79 (XSS):

1. Feature engineer extracts and normalizes all input features
2. Random Forest predicts "Medium" severity with 0.78 confidence
3. SHAP explainer identifies top contributing features: CVSS (0.4), EPSS (0.2), CWE (0.18)
4. Returns prediction with human-readable explanation: "Classified as Medium primarily due to CVSS score of 7.5 and low exploitation probability (EPSS 0.12)"

4. Intelligence Enrichment Engine

Responsibilities:

- External Data Integration: Fetches and caches threat intelligence from NVD, KEV, EPSS, and GitHub
- Data Augmentation: Adds contextual information to each vulnerability record
- Offline Capability: Maintains local cache for air-gapped operation

Key Features:

- Multi-Source Aggregation: Queries multiple APIs in parallel for speed
- Rate Limiting: Respects API quotas with exponential backoff
- Cache Management: Stores enrichment data in SQLite with TTL-based expiration
- Fallback Strategy: Uses cached data when APIs are unavailable

Data Sources:

- NVD (National Vulnerability Database)
- KEV (CISA)
- EPSS
- GitHub PoC Search

Typical Use Case: Enriching CVE:

1. NVD Lookup: Retrieves full CVE description, CWE-89 , CVSS 9.8
2. KEV Check: Finds CVE in CISA's Known Exploited Vulnerabilities catalog - sets kev_flag=True
3. EPSS Query: Returns exploitation probability of 0.85 (85% chance of exploitation within 30 days)
4. GitHub Search: Finds 3 public PoC repositories
5. Result: Original vulnerability record now enriched with 4 additional intelligence signals that dramatically increase its priority

5. Composite Risk Scorer

Responsibilities:

- Score Fusion: Combines pattern matching confidence with ML predictions
- Fallback Logic: Provides backup scoring when pattern matching is uncertain
- Confidence Reporting: Generates transparency metrics for analysts

Key Features:

- Weighted Averaging: Configurable weights for pattern vs. ML contributions
- Conflict Resolution: Handles disagreements between pattern and ML classifications
- Uncertainty Quantification: Reports confidence intervals for edge cases

Typical Use Case: Pattern matcher found a 0.72 confidence match classification as “High”, but ML predicts “Critical” with 0.81 confidence.

1. Both sources available, neither has overwhelming confidence
2. Calculate weights: Pattern - 0.47, ML - 0.53
3. Convert to numerical: High=3, Critical=4
4. Weighted average: 3 x 0.47 + 4 x 0.53 = 3.53
5. Round to nearest: Critical
6. Final classification: Critical with justification that ML had higher confidence

Output Layer Components

1. Multi-Format Report Generator

Responsibilities:

- Report Generation: Produces vulnerability reports in Excel, CSV, JSON, and PDF formats
- Prioritization: Sorts vulnerabilities by composite risk score
- Customization: Applies HighViz formatting standards

Key Features:

- Excel Primary Output: Columns for severity levels, risk scores, etc.
- JSON for Multiple Outputs: Universal data integration for multiple formatting processes
- PDF Summaries: High-level overview for clients

Typical Use Case: After processing 5,000 vulnerabilities:

1. Generate Excel report with top 500 vulnerabilities prioritized by risk
2. Export full JSON dataset for integration with Jira ticketing system
3. Create PDF executive summary showing:
    - Total vulnerabilities: 5,000
    - Critical: 42 (0.8%)
    - High: 287 (5.7%)
    - Medium: 1,823 (36.5%)
    - Low: 2,848 (57.0%)

2. Local SQLite Database

Responsibilities:

- Storage: Stores all vulnerability records, enrichment data, and analyst feedback
- Caching: Speeds up repeated queries and enrichment lookups
- Logging: Maintains comprehensive logs of all system operations/outputs

Key Features:

- Logging Folder: Folder of all recent logs from different system scan
- Cache Expiration: Enrichment cache invalidated after several days
- Indexed Queries: B-tree indexes on plugin_id, cve, timestamp for lookups

Typical Use Case: When processing a new scan:

1. Check if PluginID exists in vulnerabilities table
2. If yes, retrieve previous classification from classifications table
3. Check cache enrichment for recent threat intelligence
4. If cache miss, fetch from APIs and store in cache for future scans
5. Log all operations to log file for traceability

Architectural Styles and Influences

1. Pipeline Architecture

The system embodies a sequential data processing pipeline where each stage transforms data before passing it to the next:

- Stage 1: Parsing – Parsed records
- Stage 2: Enrichment - Enriched records
- Stage 3: Classification - Risk scores
- Stage 4: Reporting - Outputs

Benefits:

- Modularity: Each stage is independently testable and replaceable
- Scalability: Stages can be parallelized or distributed
- Clarity: Data flow is linear and easy to understand

2. Layered Architecture

The three-layer design follows separation of concerns:

- Input Layer: Deals only with data acquisition and normalization
- Processing Layer: Contains all logic and intelligence
- Output Layer: Handles presentation

Benefits:

- Maintainability: Changes to output format don't affect ML logic
- Testability: Each layer can be individually tested during unit testing

One of the major differences between the original plan for this product and what we ultimately created was the focus of the product. Originally, the main goal was to identify false positives and false negatives from the Nessus scans, so that the real vulnerabilities in the scan could be focused on, while the less sever vulnerabilities would not waste precious research time. This was what we had been building on for a large chunk of our production, when we had a meeting with our client to discuss where we were in development. That was when they commented that they would prefer if we focused on mapping unknown vulnerabilities, so these vulnerabilities were a main source of concern, because they don't know how severe it might be until they spend hours researching and analyzing vulnerabilities. This created a shift in our focus, where instead of looking for false positives/negatives, we instead developed the architecture around researching unknown vulnerabilities. Another minor difference between how we initially planned the project and how we developed it was the idea of using the cloud for model synchronization. Initially we saw value in allowing HighViz team members to sync any model changes with the cloud so that every team member was on the same model, it became clear that this wasn't nearly as important, mainly because the small size of the HighViz team meant that it wasn't too difficult to sync everyone together. Some changes that were added compared to our original plan include the ability to cache data from the database lookups, in case an internet connection is unavailable. This feature was not considered initially, because HighViz works primarily on local machines in their offices, so it would be unlikely that the feature would be useful. However, due to advice from our mentor, we decided to add it to the architecture as a safety measure in case the databases that the product looks for are unavailable. However, through all these changes, our product still achieves its primary goal for HighViz: it saves them valuable time during their lengthy research phase.

<u>Testing</u>

Our testing strategy included three main phases: unit testing, integration testing, and usability testing. These three testing phases allowed us to ensure not only that the system functioned properly, but also that all our requirements were met to a degree that we were happy with. Due to the nature of our product being a command-line interface, usability testing was not as important. As such, most of our testing efforts were focused solely on the unit testing and integration testing phases. Now, we will go more in-depth into each phase to detail how we used each phase to ensure our product was in working condition.

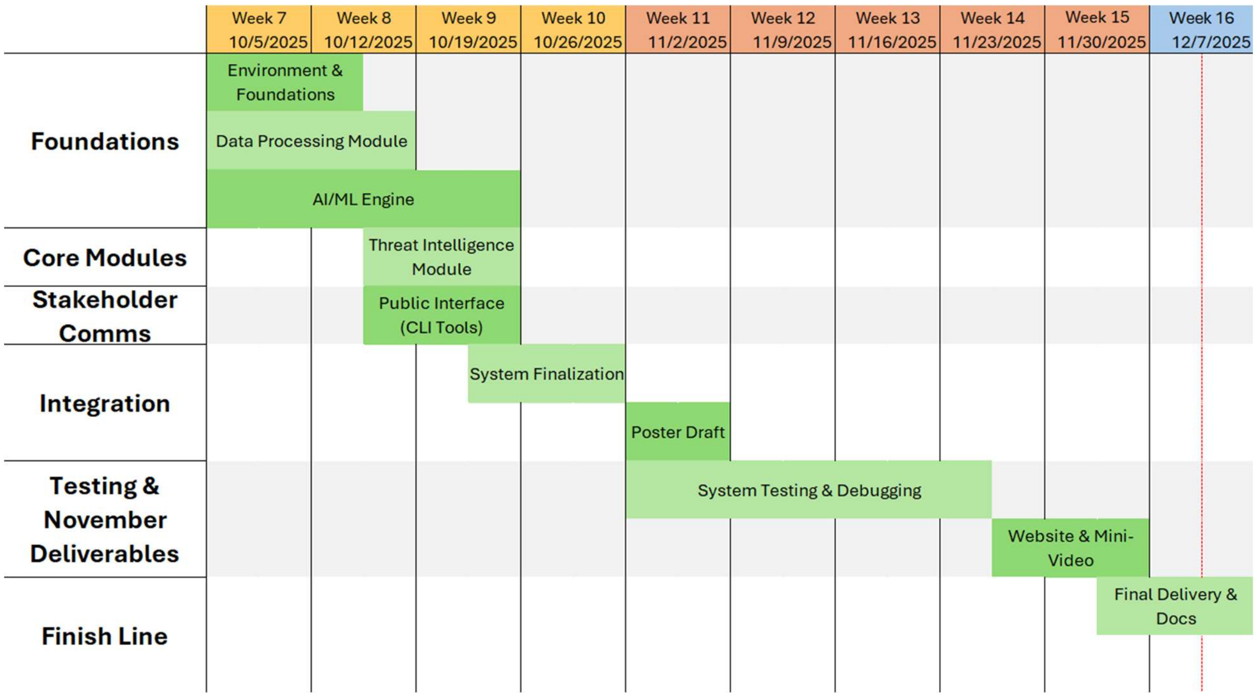
Starting with the first phase, unit testing. This was our foundation, as without it, we would have no idea whether all our modules worked individually. We would test each module isolated from each other, making sure that the module does exactly what is expected of it, without any hiccups. Whenever we had a unit test fail, that means our foundation was cracked, and we needed to immediately fix the issue before continuing with operations. This was our first line of defense when it came to testing, but that is not the only testing that needs to be done in order to make sure everything was in working order.

The second phase, integration testing, was even more vital than unit testing. This is because it ensures not only that each module worked as intended, but that when these "functional" modules are put together, they continue to work as intended. This was our focus, as when building an AI/ML product such as this, any one mistake in the chain of modules can result in a catastrophe. For example, even though our unit testing would show that our AI score would be computed with no errors, when integrating this module with other modules it was revealed that the score was being computed incorrectly and as such resulted in the entire series of modules not producing the intended result. Using integration testing to find this issue, we were able to identify the problem when creating the scoring and fix it so that the modules would function correctly when put together. However, while this was the most important phase of testing, there was still one more testing phase to go through.

Usability testing, while not a major focus for our project, was still important in making sure that everything was easy to use. After all, making sure everything works perfectly is great, but if a new user can't understand how to use the product at all, it would be all for nothing. So, once we had a product that could be demoed, we would use it ourselves, showcase it to our team mentor, and demonstrate it to our client, to make sure that someone less familiar with the product than we were could still use it to its full potential. This testing revealed some potential issues with how we chose to have our product used, such as complicated command names with even more complex arguments, when we could instead run a single, easy-to-use command with user friendly file selection and argument usage. Without this testing, we likely would have shipped a product that, while it worked as intended, would feel clunky to use and make effectively using it a challenge.

Project Timeline

When it comes to the overall project's timeline, it's important to not only outline what went on in these weeks, but more specifically, who did what. Below is a Gantt chart that goes over the project's overall production but has significantly changed in the actual process of development.



The specific portion of this graph that is not represented in the actual development of the product solution is how long it took to create the AI/ML engine. After each week, our team experienced large refinement and clarifications that our model needed to have changed. There were many hurdles in this process. The person who led in the development of the model was Christian, who did most research and delegate specific tasks. These included changing our model training data selection, de-duplicated entries that showed up multiple times, changing our model folds, changing the model to be an ensemble voting system, and finally using SMOTE to create different types of synthetic data for underrepresented data. These model changes did not last from Week 7 to 9. This lasted from Week 7 up to Week 16. Sure, this could have counted for System Testing & Debugging, but with the amount of work that needed to be implemented as well as the fact we were doing multiple full-model resets, it's fair to say that the team at Cyber Recon went through a lot of processes to integrate a fairly accurate model.

What's also not shown here is the amount of documentation that needed to be submitted and developed. Most, if not all, of the Gantt chart elements are developmental material, and not any documentation. Here, we've had two Design Reviews, a Software Testing Plan, a Poster presentation, a mini video, a final presentation, a user manual, and a Final Project Report, which you're reading now. All project documentation was led, delegated, and refined by the Team Lead Zachary Garza and developed by the full Cyber Recon team.

Future Work

There indeed were some pieces of the project that were expected to be implemented, deemed out of scope, and thus were not implemented. These were large expectations the client (and the Team) had for the project but of course were not deemed as necessary. Such requirements included multiple output formats, full offline functionality, and docker implementations. Many of these functionalities were not deemed as important or part of the normal user flow, so it ended up being adjusted massively and were dropped.

However, there were some non-functional requirements that ended up being implemented. Some of these things included updating the map file, which the team was happy enough to make one of the main showcases of the product, as well as research on many different places to make the vulnerabilities more enriched.

Regarding the actual Future Work of this project, there were a couple pieces of information that were tossed around as "to-be" implemented but were not due to not being in the project scope. One of these things was the application of automatic pushes and pulls to project directories once a model has been retrained. At that moment, we said it wouldn't be entirely difficult to go through with this, but as it required a lot of work on their end with repository settings, branch permissions, ect., it became more of an issue that our client could solve, especially since it was out of the scope of the project. Another aspect we tossed around briefly was the idea that our AI engine was to use ChatGPT. This would have completely thrown the entire project for a loop and be completely different due to the security portions. We did not implement this due to requirements having to be completely different.

Lastly, there was a glaring implementation that the client wanted. After many meetings with the client, we identified that their workflow not only involves researching but creating a report to go back to the customer. Another major time save that the client could use would be to create a report for their customers directly from our scan report. This would require some easy data-frame mapping, but it could be done and be in our client's format. Obviously, this was not in the expected product requirements, so it wasn't going to be worked on. If this were part of the requirements, this would make the product more challenging but increase the product's value.

There is also room for discussion regarding how to make the product improved, rather than attaching new features to it. All of these are, again, out of scope of the project and not part of its requirements, meaning it was not going to be implemented. On top of the main support for Mac, additional functionality for Windows machines could be implemented as well. This could improve the possible amount of workforce that our client could have in their current workflow. Another improvement could be to reduce the number of files in the git repository, but this has no actual impact on the functionality of the product and rather assists with maintenance.

Conclusion

Overall, we deem this project a success. Our client's prior workflow involved using the Nessus scanner to get testing data from multiple different applications, websites, and machines of their customers, in which they have their own workflow to generate a report that is in a human-readable format. They then go through their report and look at every single unknown vulnerability and do their own research. This process can be anywhere from 30-60 minutes for each unknown vulnerability. This means that each client can take hours or even days of time simply just researching each unknown vulnerability. Obviously, to save people from the threat of cyber security, time must be saved somewhere. This is where our team has come in to handle this problem for them. Our client not only requested that research is done for them but has also an AI assistant that attempts to replicate a severity score for vulnerabilities just as the client would do.

Specifically, what we built ended up satisfying all these requirements. We built…

- An AI Tool that helps to replicate what our client would say regarding the severity of unknown vulnerabilities using a map.
- Researched all unknown vulnerabilities, now even researching vulnerabilities already known.
- Significantly improved the time it takes to research unknown vulnerabilities.

While this solves an issue for our client, it could very well be applied to all sorts of businesses out there that are attempting to remain cyber-secured. This will help people who aren't well versed in cyber security to assist in other parts of analyzing. This impact can assist more people to be successfully cyber secured on the internet.

Glossary

- **CVSS:** Common Vulnerability Scoring System
- **CVE:** Common Vulnerabilities and Exposures
- **KEV:** Known Exploited Vulnerabilities (by CISA)
- **EPSS:** Exploit Prediction Scoring System
- **BERT:** Bidirectional Encoder Representations from Transformers
- **ML:** Machine Learning
- **NLP:** Natural Language Processing
- **CLI:** Command Line Interface
- **UI:** User Interface
- **Nessus:** A vulnerability scanner used to get a list of vulnerabilities with supporting information.
- **SMOTE:** Synthetic Minority Over-sampling Technique
- **CWE:** Common Weakness Enumeration

Appendix A

This appendix explains how a new team member can fully set up their machine to develop, test and produce new versions of the Cyber Recon system. It walks through the same learning curve that we had at the beginning of the project, covering hardware, software tools, environment setup, and the complete production cycle from editing code to generating updated reports.

Hardware

Our team primarily developed on macOS (Apple Silicon). These machines had 8 GBs of RAM, 4 CPU cores, and at least 5 GB of available disk space. The system runs entirely on CPU, so no GPU is required. A modern machine with at least 8GB of memory is recommended to comfortably process larger Nessus scans, run machine-learning components and open files in development environments without performance issues. Developers using 8 GB of RAM may need to close other applications during long run times.

Toolchain

The Cyber Recon project is built on Python and a series of supporting tools. Python 3.12 is the core programming language and runtime used for every part of the system, including the Nessus parser, reporting tools, machine learning scripts, the Tk-based file picker, and threat-intelligence integrations. Using Python 3.12 specifically is required because several native libraries and GUI components depend on that version.

Most development occurred in VS Code, which provided linting, code completion, formatting, and Git integration via extensions such as Python, Pylance, Black, Flake8, Docker, and GitLens. Some developers used PyCharm for deeper refactoring or search capabilities. Git and GitHub handled version control, branch management, and collaboration.

Setup

1. Install Python 3.12. On macOS this can be done through Python.org or through homebrew.
2. Clone the repository by running:
    a. **`git clone https://github.com/minibutler/Cyber_Recon`**
    b. **`cd Cyber_Recon`**
3. Install project dependencies by running: **`pip install -r requirements.txt`**

Production Cycle

1. Modify source code
2. Run scan.py to execute main Cyber Recon pipeline
3. Run update_map.py to run automated findings_map.csv update
4. Run python src/tools/train_from_findings_map.py to retrain model on updated findings map

This appendix provides a complete walkthrough of the hardware requirements, tools, environment setup, and production workflow used throughout the Cyber Recon project. A new team member following the steps above will be able to fully configure their development machine, modify code, regenerate Nessus-processing outputs, retrain ML models when necessary, and push updates back to the repository. With this knowledge in place, they can immediately begin contributing to the development and maintain consistency across all team members' environments.