CYBERGATORS

A dynamic software application to determine the holistic cyber resiliency of a user-specified system under evaluation (SUE)

Ozlem Polat – Team Leader/SCRUM Master
Samson Carter – Program Manager
Jessica Lourenco – Backend Lead
Andrew Ballard – Developer
Shayan Akhoondan – Developer

Advisor: Dr. Cheryl Resch
cheryl.resch@ufl.edu

CIS 4914 Senior Project
University of Florida

YouTube Link: XXXXX

Key Words

Cybersecurity, Cyber Resilience, Bayesian Networks, Finite State Machine, Fuzzy Logic, Risk Assessment, Systems Modeling, Graph Databases, Neo4J, Python, Dash, Attack Graphs, CVE Analysis, Vulnerability Modeling

Project Overview

CyberGator is a software application developed to evaluate and improve the cyber resilience of complex information systems. Created in response to the 2024 Cyber Defense Performance Measurement Challenge (CRAM Challenge) organized by the Naval Surface Warfare Center, Dahlgren Division, this tool was designed to assess two simulated enterprise networks, referred to as Systems Under Evaluation (SUEs).

These networks model the internal architecture and vulnerabilities of a fictional technology company, MRZTech, which operates under high-value security constraints and faces risk from both external adversaries and insider threats. CyberGator provides a framework for analyzing such systems by visualizing attack pathways, simulating critical threats, and calculating resilience scores that reflect a network's ability to withstand, respond to, and recover from cyber incidents.

The software integrates several analytical approaches to simulate and assess cyber risk. It uses Bayesian Networks, which are probabilistic graphical models that represent variables and their conditional dependencies. In CyberGator, these networks help estimate the likelihood of successful attacks based on known vulnerabilities and network configurations. Finite state machines are also used to represent the possible states of each node, such as secure, degraded, or compromised, and the transitions between those states as certain conditions or attacks are applied. This structure makes it possible to simulate dynamic progress of attacks through a system.

To convert raw vulnerability data and structural information into meaningful scores, CyberGator applies Fuzzy Logic, a form of reasoning that allows for degrees of truth rather than binary judgements. This enables more nuanced assessment of cyber resilience by capturing uncertaining and partial risk, conditions that are common in real-world cyber-physical environments.

While CyberGator was originally designed to integrate with a Neo4j graph database for dynamic modeling of nodes and attack paths, limitations with database schema uploads led the team to implement a hardcoded JSON-based structure instead. This JSON file stores all system nodes, vulnerabilities, and critical functions, enabling the application to simulate and assess cyber resilience without relying on an external database engine. A Neo4J graph has been included within our application to demonstrate the future goals and a visual representation of the SUE. The front-end interface, built with Dash, a Python framework, reads from this structured data file to render attack graphs, display node-specific information, and support simulations involving CVEs and system state changes. Though Neo4j remains a planned future enhancement, the current implementation provides a fully functional and flexible assessment tool using lightweight, file-based data handling.

CyberGator serves a practical need in both academic and operations cybersecurity environments. Traditional risk assessments often fail to capture the dynamic nature of threats or the interconnectedness of modern enterprise systems. Tools like CyberGator offer security professionals, students and researchers a platform for exploring the “what if” scenarios, understanding vulnerability propagation, and evaluating mitigation strategies in a controlled, testable way. As digital infrastructure continues to grow more complex and attackers become more sophisticated, employing tools like AI penetration, being able to simulate and quantify resilience in advance of a real compromise is an essential step toward building systems that are not just secure, but adaptive and recoverable.

Deliverables

Completed Deliverables

Our current project is a Dash-based User Interface with multi-page navigation for system nodes, CVEs, critical functions, environmental factors, and export tools.

JSON/CSV Data Model including:

- Nodes_Complete.json: Primary system representation
- Critical_Functions.json, Fuzzy_Set.json, Attack_Tree.json: Used in resilience scoring
- software_inventory.csv and software_cves.json: Software metadata and CVE tracking

Simulation Engine with the ability to:

- Add and remove software CVEs per node
- Lock/unlock environmental risks
- Modify critical function mappings
- Recalculate resilience scores using fuzzy logic and finite state machines

Output Generation including:

- Resilience_Scores.json, Individual_Node_Metrics.json, and System_Resilience_Scores.json
- Export to CSV functionality

Other completed components of the project are:

- Authentication System with login, signup, session, and logout logic implemented and unit-tested.
- Data Loader Module to manage reading, writing, and resetting system data.
- Dockerized Deployment with Docker Compose and Poetry integration.
- Testing Suite for authentication services and controller logic using unittest and mock.

- Architecture Diagram and supporting documentation.

Functional Specifications

- Input Handling: JSON and CSV-based configuration files editable within the app
- Output Handling: Exportable scoring and metrics in JSON and CSV formats
- Resilience Modeling: FSM and fuzzy logic-based resilience scoring
- User Controls: Add/remove nodes, edit CVEs, view attack trees, export results
- Reset Function: Manual restoration of simulation data from backups

Current Status

CyberGator is a functional prototype that supports simulation of advanced persistent threats, visualization of attack paths, and resilience scoring using local file-based data. The core simulation and user interface are complete and operational. All primary use cases—CVE injection, score generation, system export—are implemented and manually verified. Authentication services are working but not integrated into the UI.

Remaining Work and Backlog Items

1. Resilience Model Accuracy: The scoring engine is unvalidated and requires further research to assess accuracy.
2. Neo4j Integration: Postponed due to import schema limitations. JSON remains the default data store.
3. Reset Functionality in UI: Current reset process is manual; a user-triggered UI reset is desired.
4. Usability Enhancements: UI still requires polishing, improved responsiveness, and error handling.
5. Simulation versus Live Mode: toggling between simulation mode and live mode to persist changes within the system versus making permanent changes.

6. Audit Trail for Live Changes: We wanted to integrate a tool to track the changes made within the system to demonstrate the changes in the system based on individual changes over time and how they combine. Currently, we can export system information at any given time, but not track the changes made
7. Simulation Hypotheses & Validation: A planned feature will allow users to record hypotheses before making simulated changes—predicting the expected impact on resilience scores. This feature is under development and is intended to support deeper analysis and critical thinking during simulation planning. The system will then:
    - Log each change and score adjustment
    - Compare predicted outcomes with actual results
    - Highlight matched vs. deviated predictions with magnitude details
    - Generate an exportable report of all hypotheses and their results
8. Historical Resilience Score Tracking: A planned feature would allow users to view and compare resilience scores over time using visualizations and tabular summaries. The system would store snapshots after key simulations or updates, highlighting trends and anomalies to support longitudinal analysis. This feature remains unimplemented.
9. AI-Based Resilience Recommendations: A conceptual feature was proposed to integrate AI-driven recommendations that analyze the system's current configuration and suggest the most impactful changes to improve resilience. This functionality is not yet developed and remains in the ideation phase.

Technical Details

Codebase Overview

The CyberGator application is hosted on GitHub at https://github.com/OzPol/cybergator. The codebase is structured into distinct directories for views, services, models, database integrations, and static data, with

a clear separation of concerns across the front-end interface, back-end services, and data handling components.

Development followed a feature-branch workflow. Team members created branches to address specific tasks or sprint goals, then submitted pull requests to merge into the main branch. All merges were subject to code review by at least one peer and were validated through a GitHub Actions-based integration testing pipeline. This automated CI system ensured that each merge passed basic functionality and regression tests before being incorporated into the live codebase.

Rather than maintaining long-lived named branches, short-lived branches were created per feature or sprint cycle and discarded after a successful merge. This approach encouraged clean commit histories and minimized merge conflicts. All work was tracked in the team's Jira backlog, with branches linked to user stories for traceability.

Version control was handled entirely through Git and GitHub, with team members committing frequently as they worked through defined components of the application. GitHub Actions enabled consistent enforcement of standards and testing prior to merge approval.

Key Dependencies and Libraries

CyberGator is built using a combination of Python-based libraries and frameworks, tailored for full-stack development, data modeling, and simulation. The front-end interface is implemented using Dash, a reactive Python framework for building web applications, with Plotly for rich data visualizations. Pandas is used extensively in both the front and back ends for structured data manipulation, tabular processing, and scoring calculations.

The back end is powered by Flask, which provides routing and application logic support, and NumPy, used for numerical operations and array-based simulations. JSON files are used as the primary medium for storing system configuration and simulation state, with a modular design that separates raw input, working copies, and output results. While a PostgreSQL database is present to support Neo4j integration,

the current application operates entirely from a hardcoded JSON-based data layer, as Neo4j schema issues prevented full implementation.

CyberGator uses Docker for containerization. The application is packaged in a Docker environment to ensure consistent deployment across systems. Though not directly integrated into every development task, Poetry is included in the backend for dependency management and environment configuration. It is used to define and install the required libraries and may support virtual environment isolation, though its exact role would benefit from clarification by the backend lead.

Deployment Instructions

To run the CyberGator application in a local development environment, the following tools must be installed:

Required Tools:

- Docker
- Docker Compose
- Python 3.12+
- Poetry

You can verify that these dependencies are installed by running:

docker --version

poetry --version

Steps to Run CyberGator (Development Mode):

1. Clone the Repository

   git clone git@github.com:OzPol/cybergator.git

```
cd cybergator
```

2. Create an .env File

Place your .env file in the project root (same level as the Dockerfile). This should include any environment variables required by Flask or the database connection. An initial .env file has been included in the application repository, make adjustments as needed.

3. Build and Launch the Application

Ensure Docker Desktop is running, then execute:

```
docker-compose build
docker-compose up
```

Once the container is running, open http://localhost:8000 in your browser to view the application.

4. Shutting Down the Application

To stop the application when you're done:

```
docker-compose down
```

5. Managing Dependencies (via Poetry)

To install new Python packages inside the container:

```
docker-compose up -d
docker exec -it cybergator_app poetry add <your-package-name>
docker-compose up --build
```

To clean up unused images:

```
docker image prune -f
```

6. Running Unit Tests

To manually execute the test suite:

```
docker-compose up -d
docker exec -it cybergator_app poetry run python -m unittest discover -s tests -p "*.py"
```

Database Schema and Data Migration Details

CyberGator uses a structured JSON and CSV-based data layer in place of a traditional relational database. While a PostgreSQL integration was planned to support Neo4j functionality, the current version of the application operates entirely from file-based storage, using static and dynamic JSON files to represent nodes, vulnerabilities, and simulation results.

Data Structure and Layout

The application reads and writes to a set of files located in data/json/, data/backup/, and data/csv/. The key files include:

- Nodes_Complete.json – The primary data file containing all system nodes, their properties, and current states.
- Fuzzy_Set.json – Contains membership definitions for fuzzy logic-based scoring.
- Critical_Functions.json – Maps system functions to their corresponding nodes.
- Attack_Tree.json – Represents the structure of potential attack paths.
- software_cves.json – Stores software metadata and associated CVEs.
- software_inventory.csv – Manages which software versions are installed on which nodes.

Simulation output is written to:

- Individual_Node_Metrics.json
- Resilience_Scores.json

- System_Resilience_Scores.json

These files are read by the front-end for display and exported as needed by the user.

Data Loader Functionality

A centralized data loading module (data_loader.py) is responsible for:

- Reading input files and returning structured data for the application.
- Writing updated simulation results to the relevant JSON output files.
- Resetting the system to its original state by copying from the backup version of Nodes_Complete.json.
- Providing dropdown menu options, filtering nodes and software by type or category.
- Ensuring data consistency during interactions with software inventory and CVE metadata.

Data Persistence and Manual Reset

Data persists between sessions unless manually reset. Any changes made during simulation—such as adding/removing CVEs or altering node configurations—are written to the output files and retained on subsequent loads. To revert the application to its original state, developers currently delete or overwrite the affected output files manually in the development environment. A reset utility is included in data_loader.py to simplify this process for node-level data.

While the application does not currently use database migrations in the conventional sense, the modularity of its file-based architecture allows for easy tracking, replacement, and validation of component-level data, supporting a flexible simulation environment during development and testing.

Software Architecture

CyberGator follows a modular, service-oriented architecture designed to separate data processing, simulation logic, and user interaction layers. The application consists of three primary components: a

Flask-based backend, a Dash-powered front end, and a JSON/CSV-based data management layer. While Neo4j and PostgreSQL integration were scoped for the initial project design, the current release uses hardcoded data files for all processing and persistence.

Front-End Architecture (Dash and Plotly)

The front-end is built using Dash, a Python framework for web-based interactive dashboards. It renders user interface components such as dropdowns, graphs, tables, and buttons, while responding to user input in real time. Plotly is used to visualize resilience metrics through pie charts, bar graphs, and simulated attack trees. The layout is organized into pages for viewing and editing system nodes, environmental factors, CVE simulations, fuzzy logic inputs, and export options.

User actions (e.g., adding software, modifying CVEs, toggling environmental conditions) trigger Dash callbacks, which communicate with back-end functions to update the data.

Back-End Architecture (Flask and JSON)

The Flask application serves as the logic layer, routing requests and coordinating interaction between the front end and the data layer. It hosts the app server and handles function calls for resilience scoring, node resets, CVE injection/removal, and node data edits. Scoring logic relies on NumPy and Pandas for numerical analysis and matrix-style operations, while file I/O is handled through standard Python libraries.

Flask also serves static files and manages any export/download routes as needed by the Dash UI.

Data Layering and Storage

Rather than using a live database, CyberGator loads all system and simulation data from structured JSON and CSV files. This decision was made due to schema upload issues with the planned Neo4j integration. A custom data_loader.py module provides a consistent interface for accessing and modifying these files. This loader handles:

- Loading system nodes and software inventory
- Resetting node data from backup
- Managing CVE metadata and critical function mappings
- Writing simulation outputs to dedicated JSON files

The application maintains separation between input (Nodes_Complete.json), output (resilience scoring files), and backup directories, allowing simulations to be re-run or reset with minimal disruption.

Component Interaction Flow

1. The user interacts with the Dash front end.
2. Dash triggers a callback, which calls a Flask route or function.
3. Flask logic processes the request, manipulates data using Pandas/NumPy, and updates files as needed.
4. Results are written to JSON output files or returned to the UI.
5. Dash re-renders the page with the updated content.

This layered design allows for future enhancements, including database migration, user authentication, and real-time scoring via integrated APIs or external data feeds. A copy of the updated System Architecture is in the appendix.

Testing Information

Summary of Testing Conducted

Testing for the CyberGator application focused on validating core application behavior, particularly the user authentication system and application build process. Unit tests were written to verify user registration, login, logout, session handling, and deletion functionality. These tests were executed using Python’s built-in unittest framework and were run inside the Docker container to ensure environmental consistency.

Authentication and session logic were tested at both the controller and service levels, using unittest.mock to simulate backend database interactions. Tests included edge cases for missing fields, invalid credentials, repeated login attempts, and attempts to delete nonexistent users.

The UI and navigation components were manually tested by the development team throughout the project lifecycle. This included confirming that Dash components rendered correctly, callback functions triggered as expected, and export/reset buttons produced the correct behavior. All file reads and writes to JSON and CSV layers were verified in manual sessions to ensure user actions were reflected in persistent output.

No automated tests were written for resilience scoring due to ongoing research into the scoring model's accuracy. As such, the simulation engine operates as a functional prototype but has not been validated against formal correctness criteria. Further testing and refinement will be required to ensure the mathematical scoring aligns with real-world cyber resiliency standards.

Test Cases and Results

Area Tested	**Description**	**Result**
Signup / Registration	Test for valid sign-ups, duplicate usernames, and missing fields	Passed
Login	Tests for valid login, invalid credentials, and already logged-in user	Passed
Logout	Valid and invalid logout sceneries	Passed
Session (/me)	Verifies session state and access control	Passed
Delete User	User deletion and handling of nonexistent users	Passed
Password Hashing	Ensures password is hashed and verified correctly	Passed
Application Build	Dockerized app builds and run in local environment	Passed

Resilience Scoring	Logic untested; model currently under development	Research Ongoing

Outstanding Bugs and Defect Log

1. Resilience scoring unvalidated: No formal test suite exists to verify the correctness of fuzzy logic or Bayesian modeling results. Score outputs are plausible but not independently verified.

2. Manual reset dependency: Resetting simulation state currently requires developers to manually delete or overwrite files in data/output/. This could be streamlined with a user-facing reset function.

3. User authentication in progress: Though tested, the authentication system is not fully integrated into the front end and remains a future enhancement.

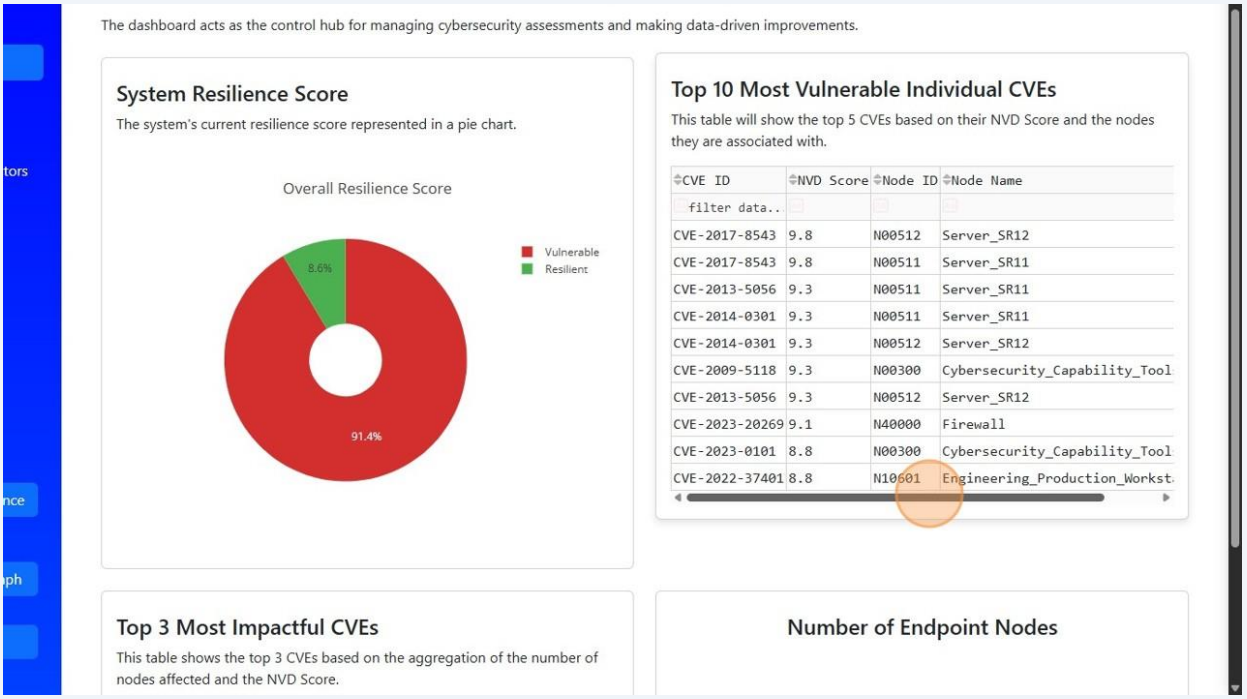
User Documentation

1 This document walks through the core features of the CyberGator platform. Users can simulate attacks, manage system vulnerabilities, and visualize cyber resilience using an interactive dashboard built with Dash, Flask, and Python. While the tool is fully functional, scoring is experimental and Neo4j integration is ongoing.

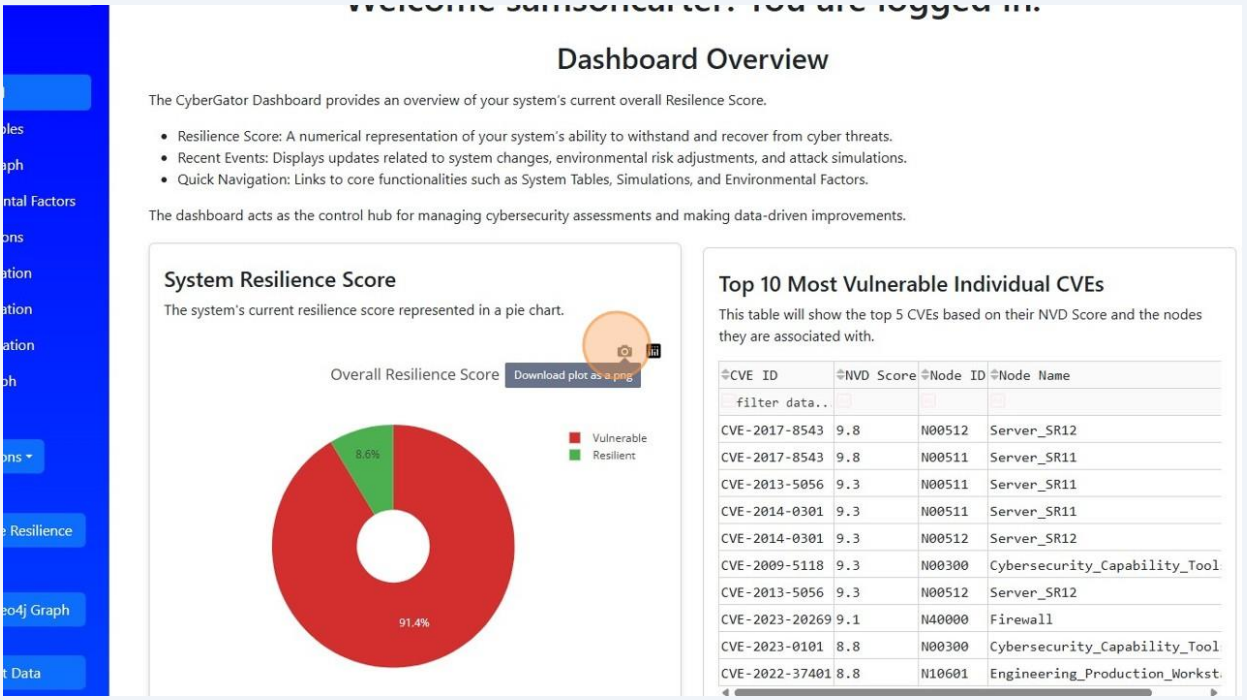
Access and Navigation

2 CyberGator runs locally via Docker. After building the container and navigating to http://localhost:8000, users are greeted with a login screen. Any credentials will work in the current build. Once logged in, you're directed to the main dashboard.

3 After logging in, users are redirected to the dashboard, where the overall score, 10 most vulnerable individual CVEs, Most impactful CVEs and Number of system endpoints are displayed



4 To download a static copy of the current System Resilience score, select this button



5

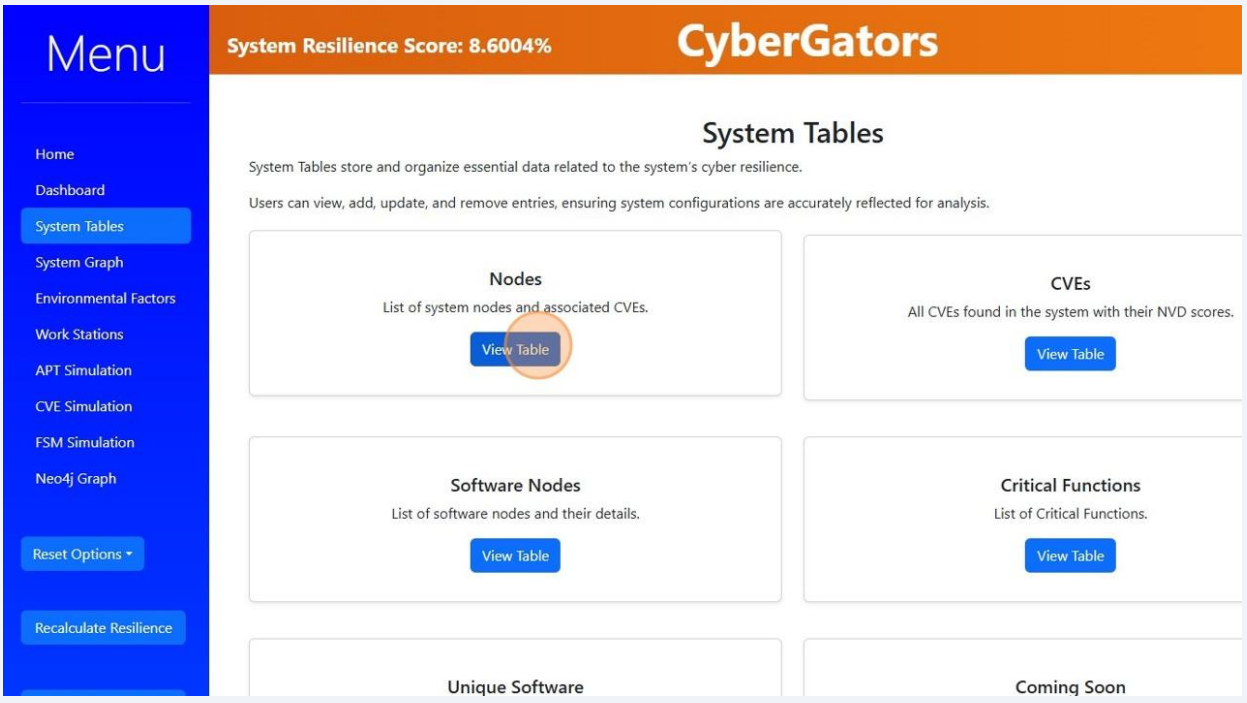
Navigation: Use the left-hand sidebar to access each functional area of the app:

- Dashboard: Overview and welcome
- System Tables: Manage nodes, CVEs, software, and critical functions
- Simulations: Run attack scenarios and score the system
- Export & Reset: Download data or revert changes

Key actions like Reset System, Recalculate Resilience, and Export Results are located at the top of each page where relevant.

System Editing and Tables

6 Click "System Tables" to view the system tables



7

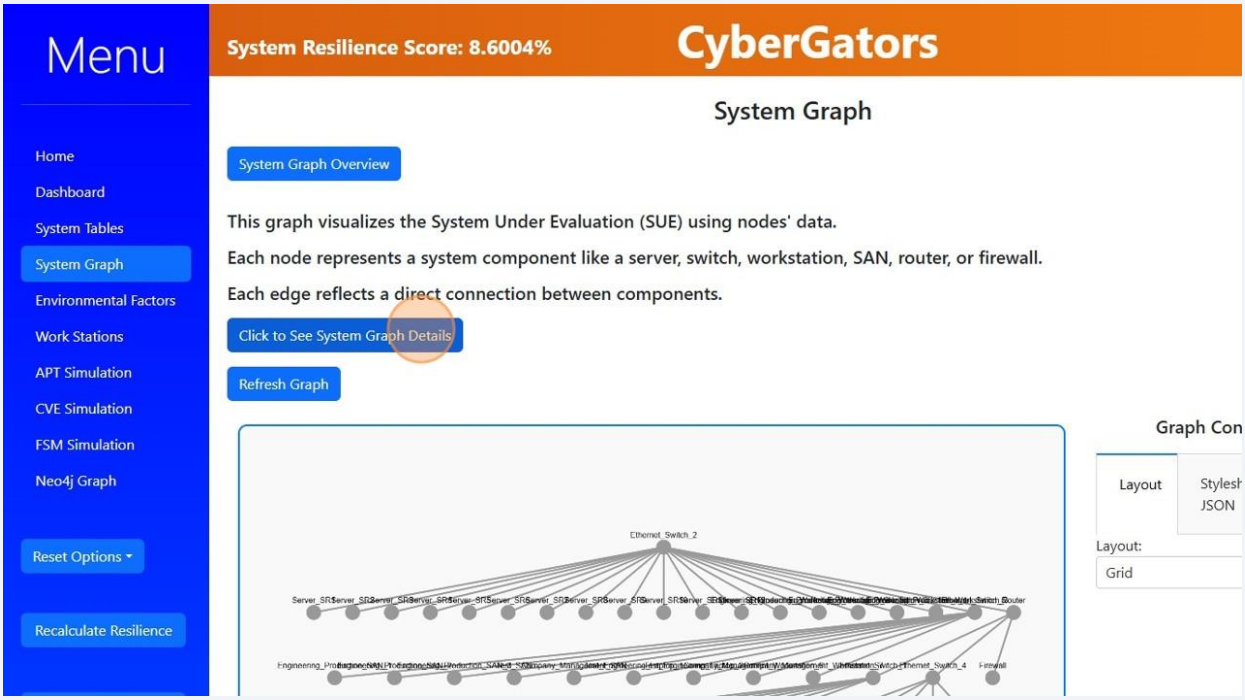
From the System Tables section, you can inspect and update the following:

- Nodes: View or edit devices and systems.
- CVEs: Inject vulnerabilities into specific nodes.
- Software: See installed packages and add/remove entries.
- Critical Functions: Link system components to mission-critical processes.

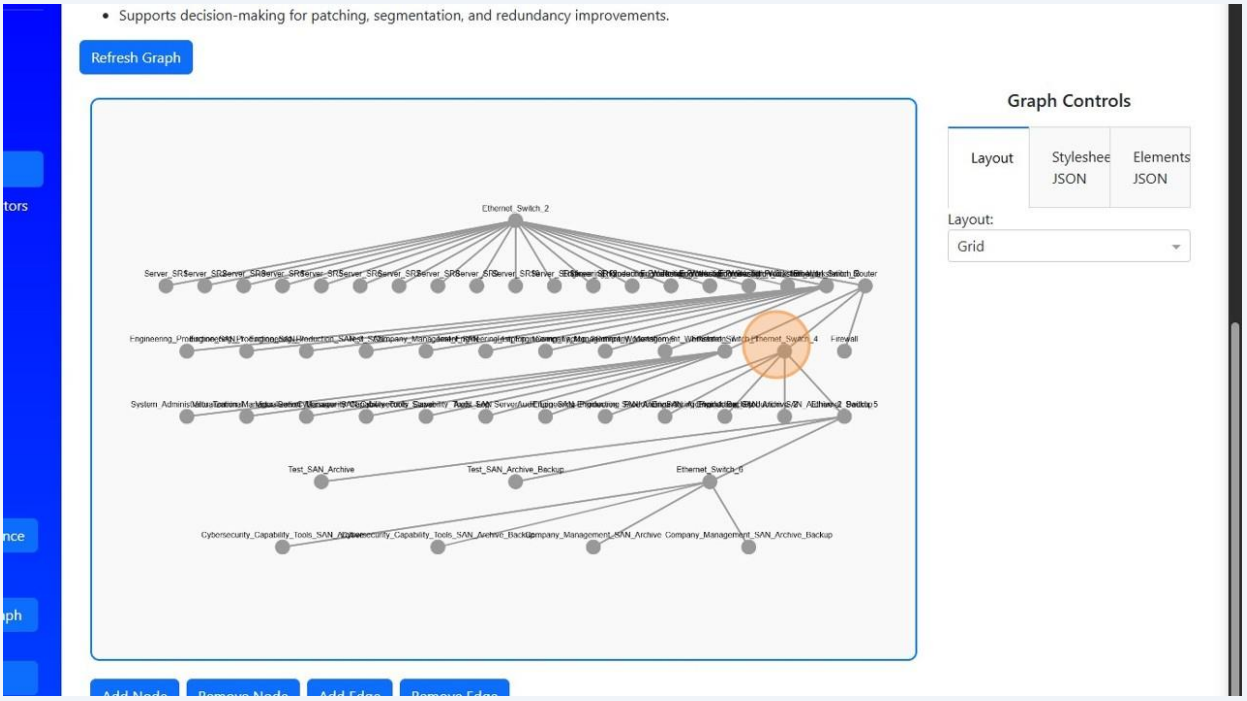
Each table supports inline editing and updates directly affect simulation outcomes.

8

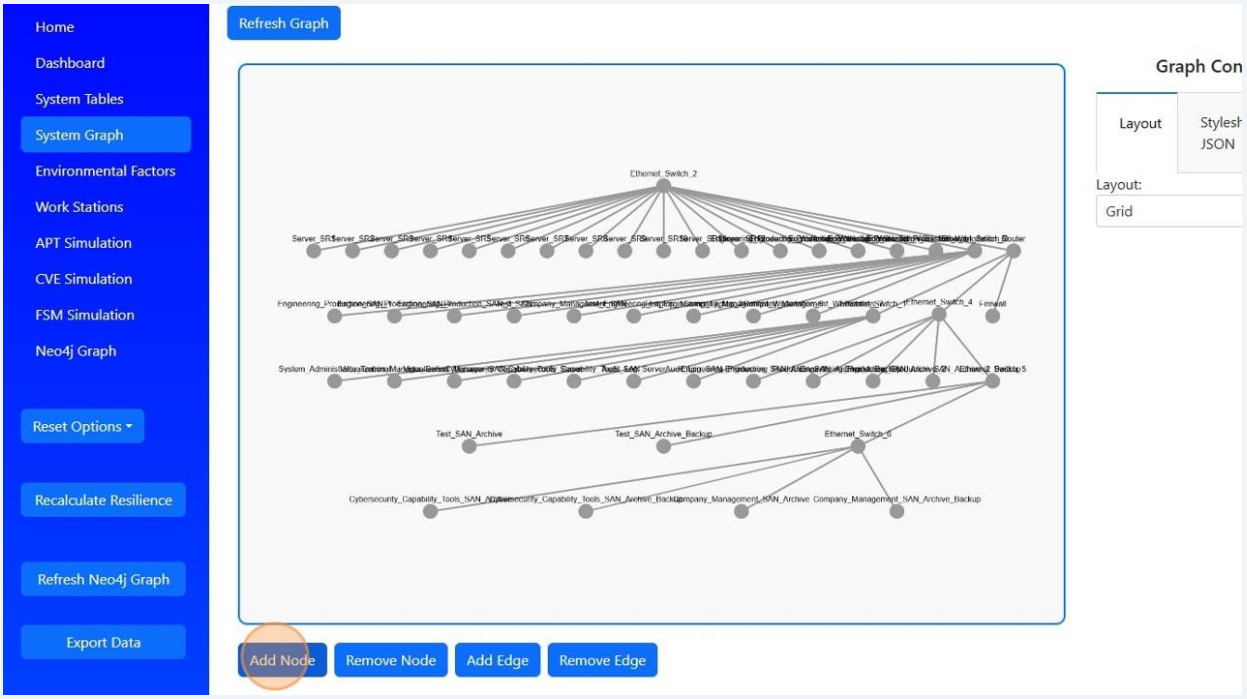
Next, let's explore the system graph table. Click "System Graph" on the sidebar.. To learn more about the system graph and what it means, click "Click to See System Graph Details"



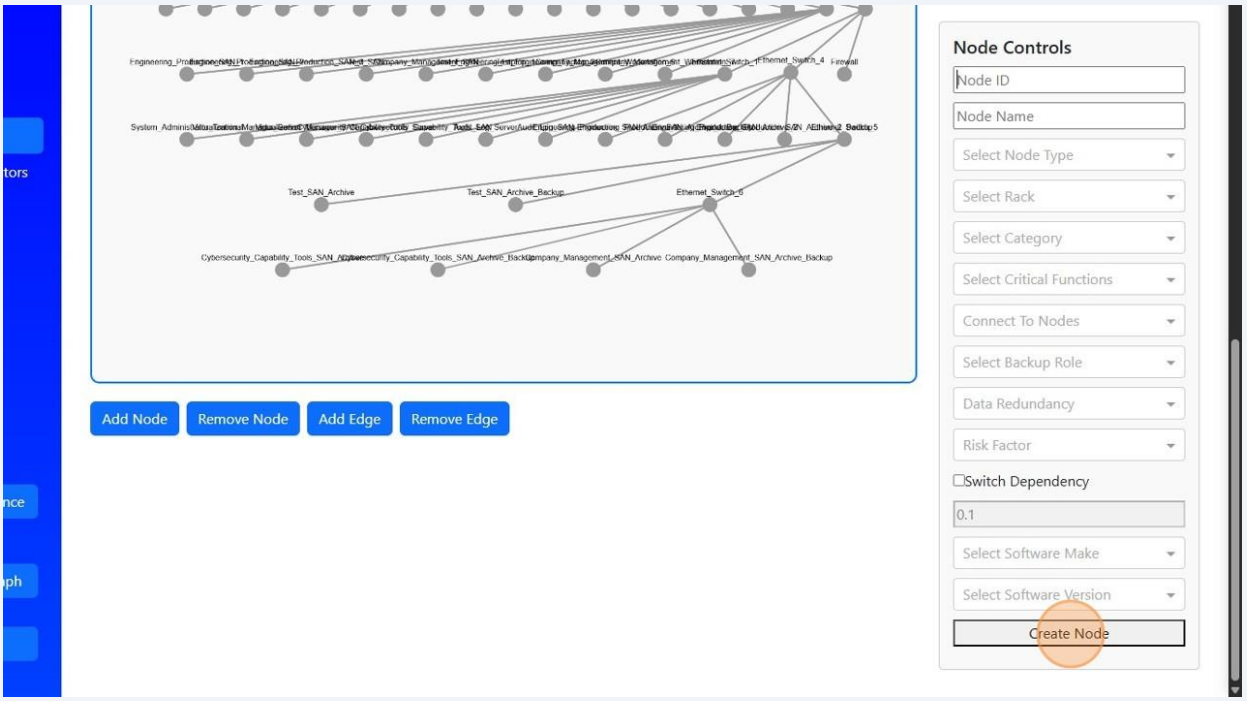
9 Select a node within the graph by clicking on it.



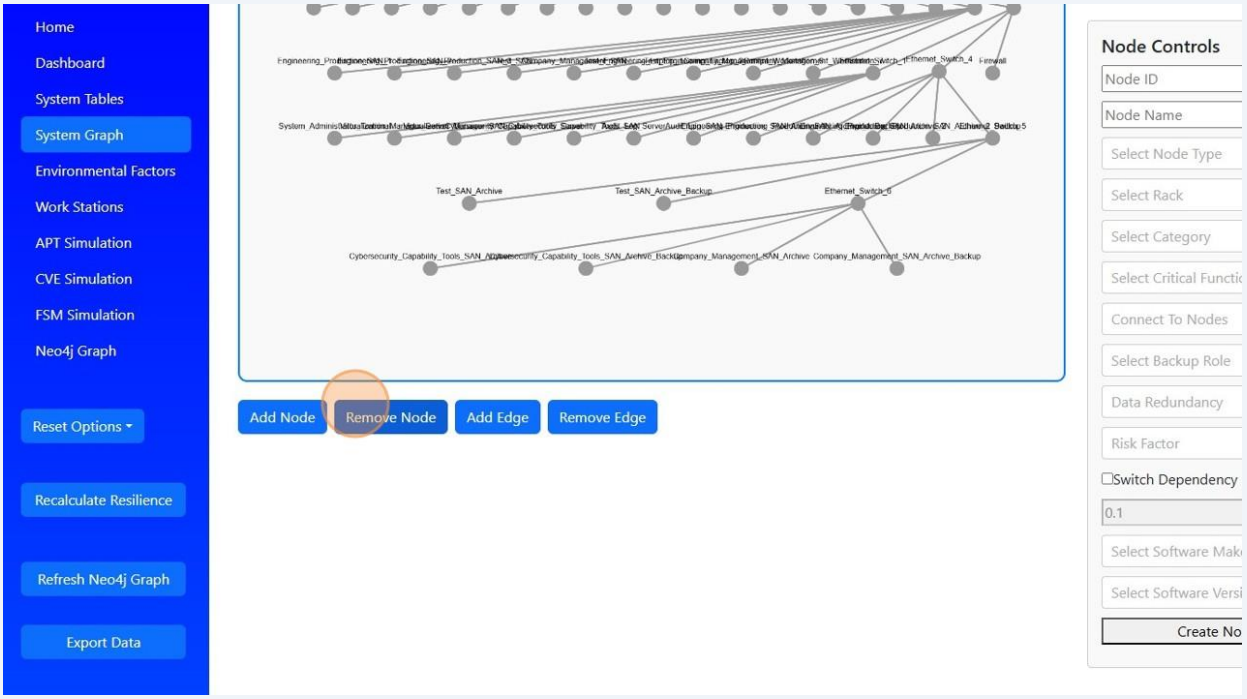
10 To add a node, click "Add Node"



11 Input the prescribed information on the right. Click "Create Node" to add the node. Refresh the graph at the top to view the new node within the system.

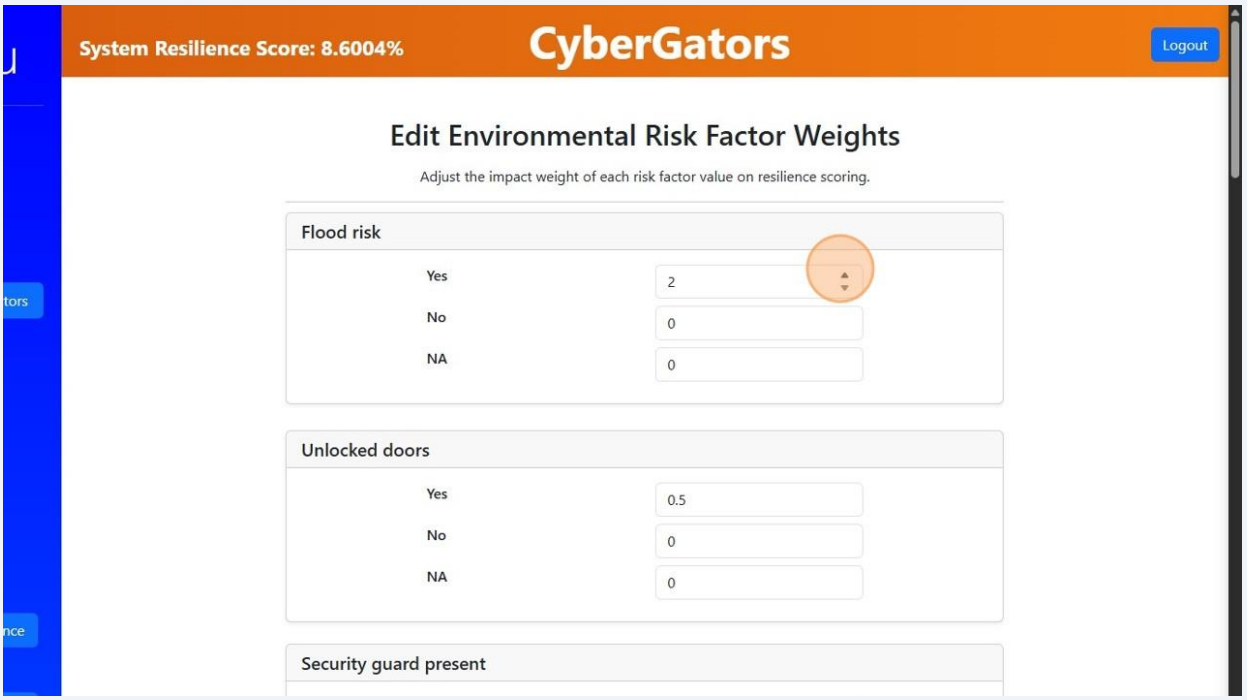


12 To remove a node, click a node on the graph, and then click "Remove Node", followed by refresh system graph.

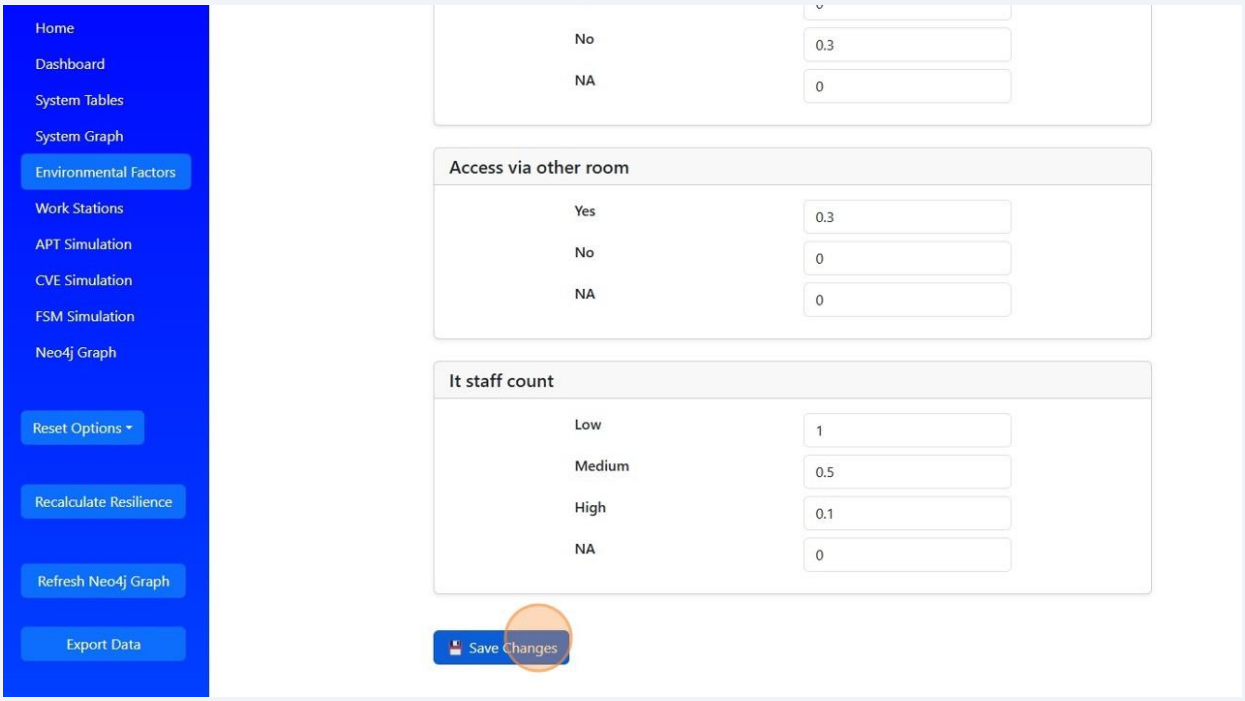


13 Edges can be added and removed by selecting a node or edge and selecting the corresponding buttons.

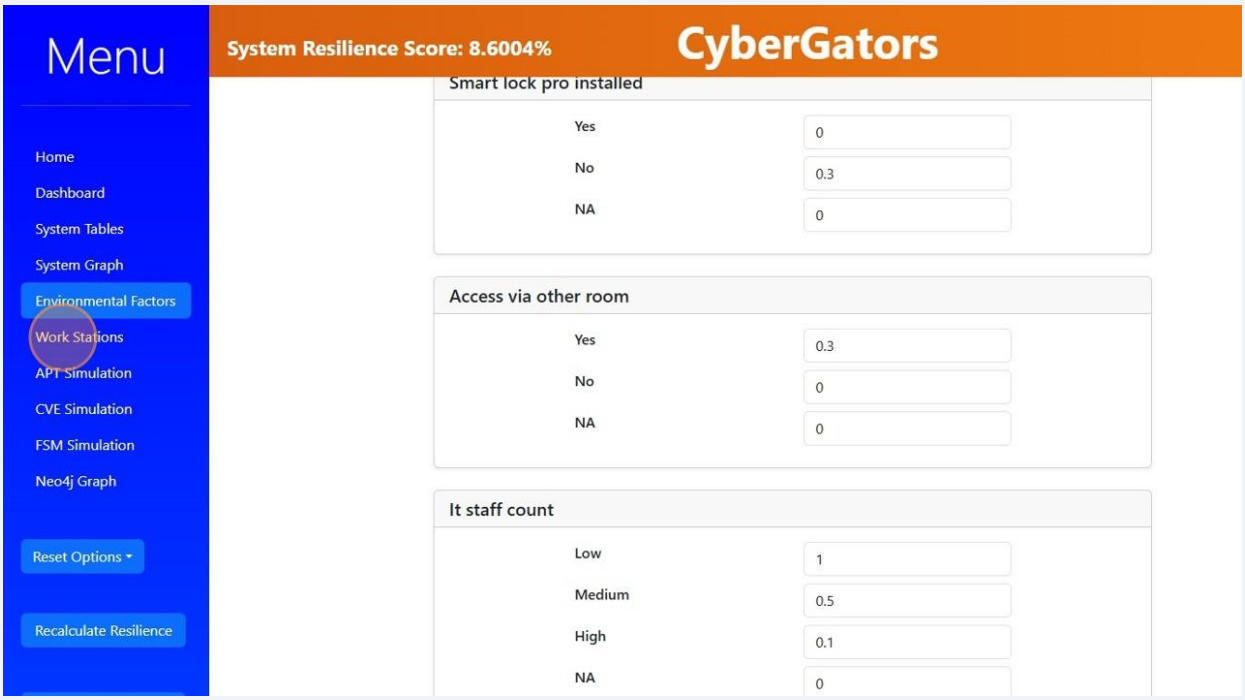
14 To view and edit the environmental risk factors, click "Environmental Factors" in the sidebar. Edit the environmental risk factors by adjusting the number field to the right of each risk factor.



15 Click "Save Changes" to save the changes to the environmental risk factors.

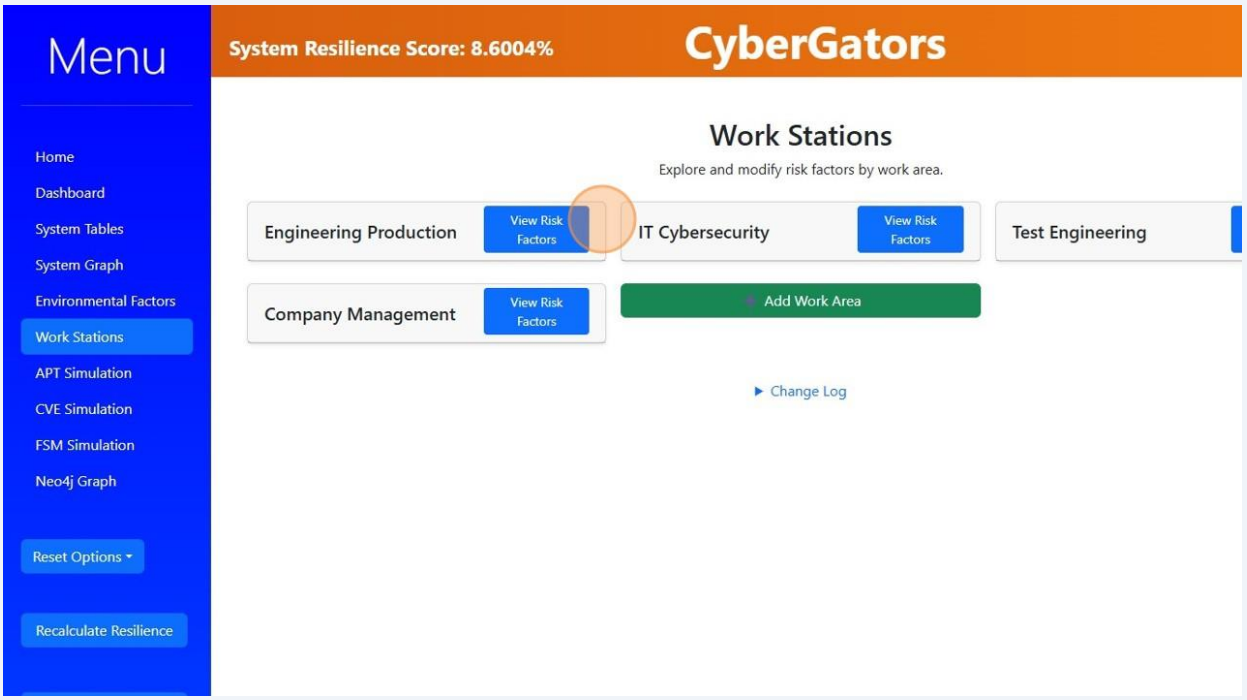


16 To view the work stations within the system, click "Work Stations"

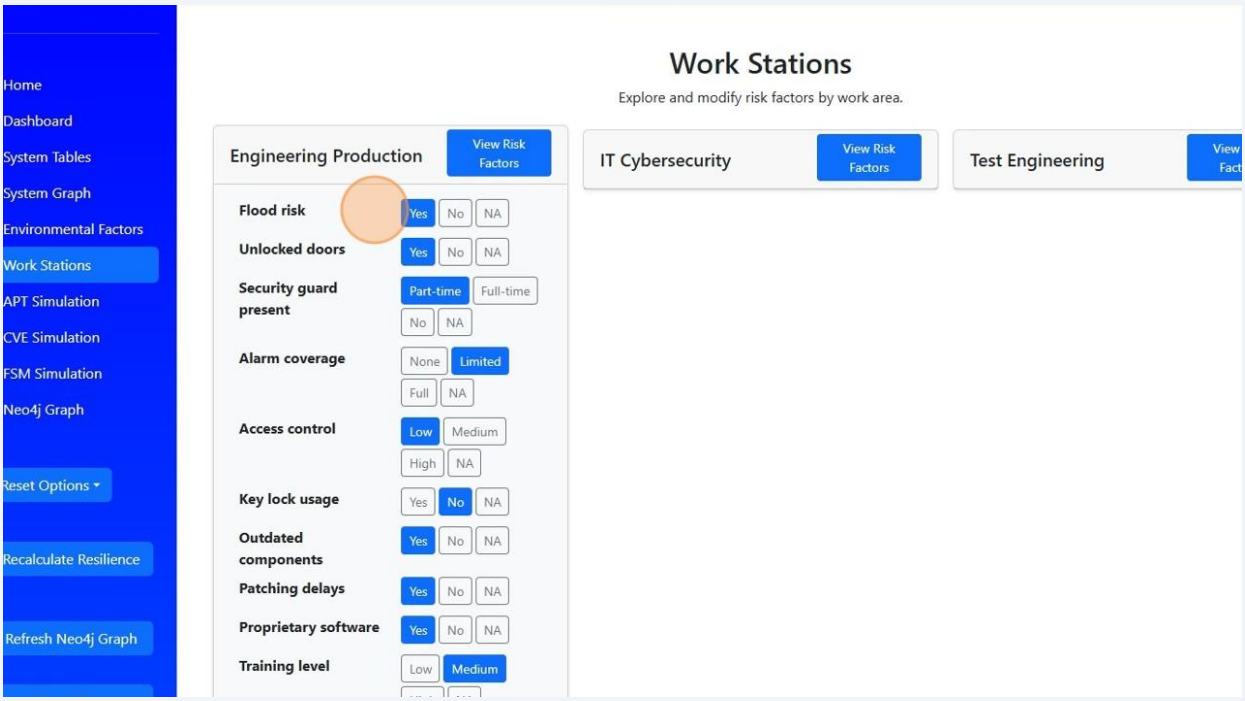


17 Click "Save Changes" to save the changes to the environmental risk factors.

18 Work stations and their associated risks can be viewed by clicking the "View Risk Factors" button by each workstation.

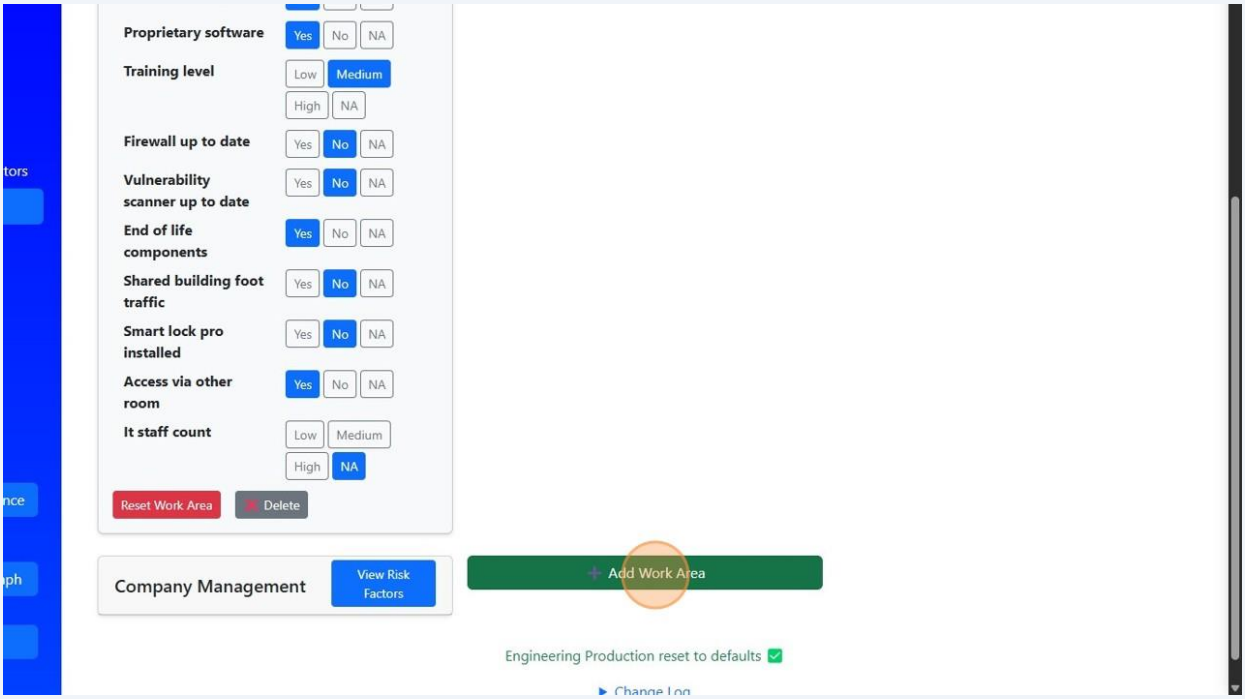


19 Make changes to the Risk Factors of a given workstation by using the scales built in to the Environmental Factors page.



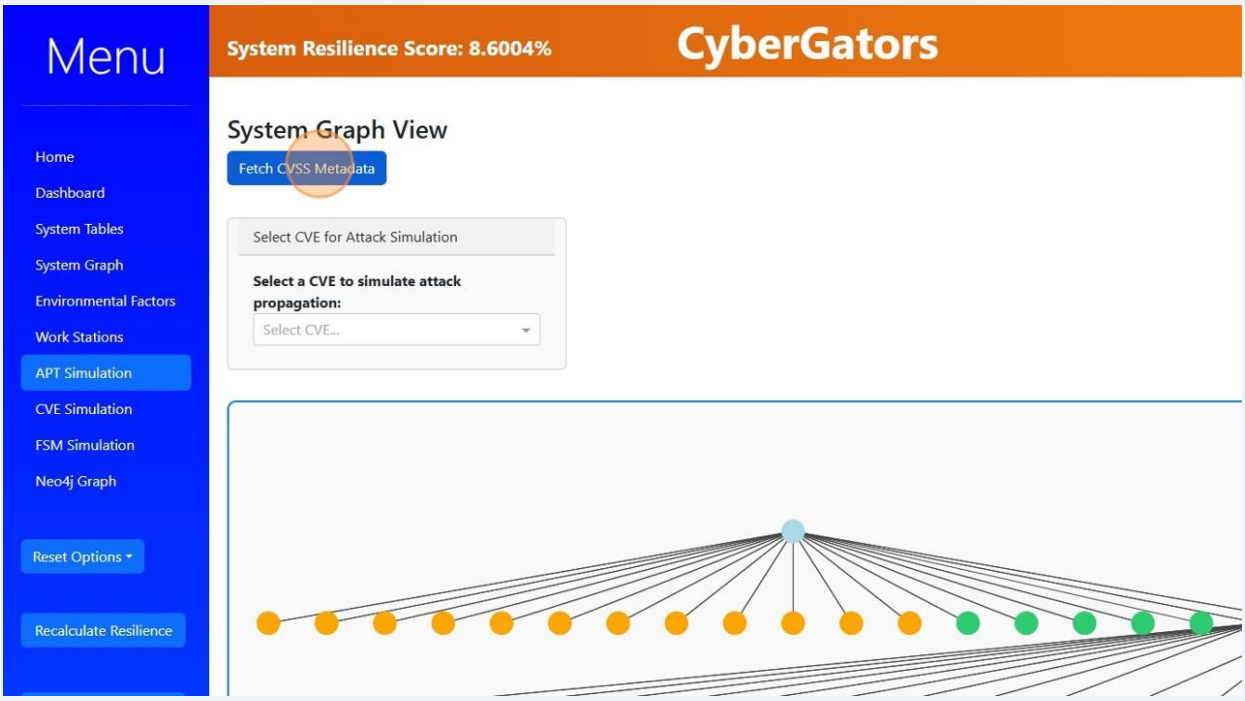
20 To reset the work area back to its initial values click "Reset Work Area". Or, select Delete to remove a work station.

21 Click "+ Add Work Area" to add a work area to the system. Give it a name and hit "Add." the new work area will be built with the same environmental risk factors as the other stations.

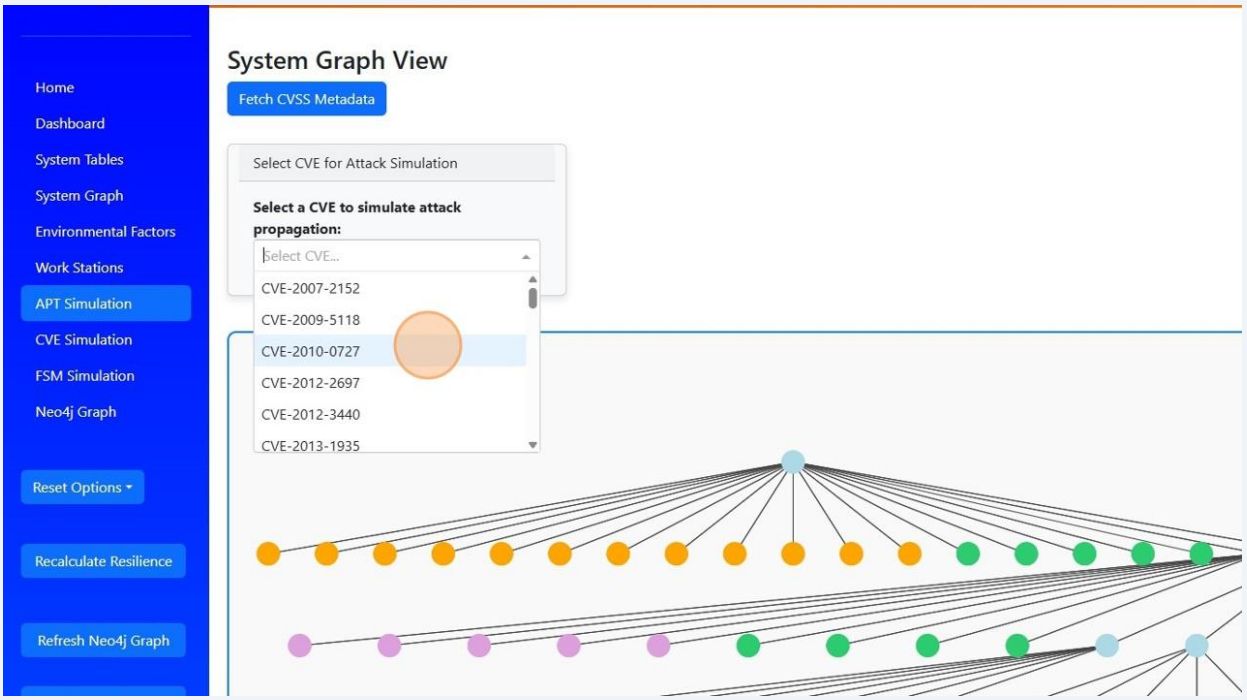


Simulations

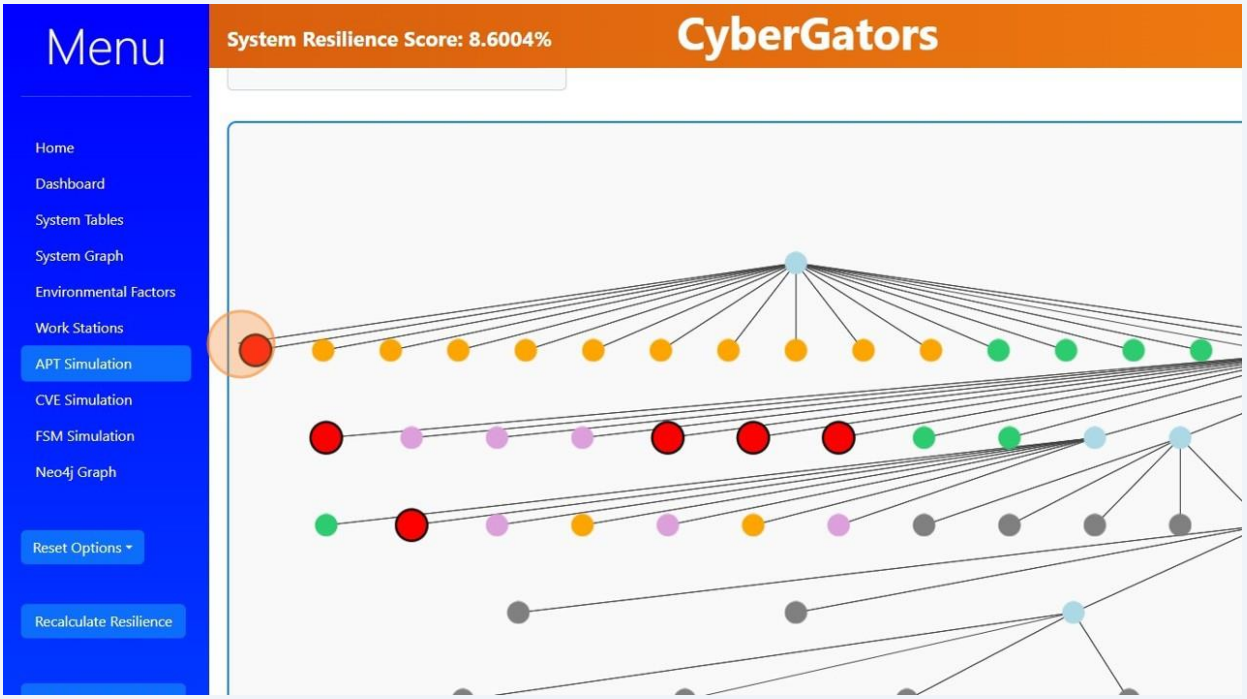
22 To view the Advanced Persistent Threat Simulation, Click "APT Simulation." Click "Fetch CVSS Metadata" to fetch most up to date CVSS metadata from MITRE.



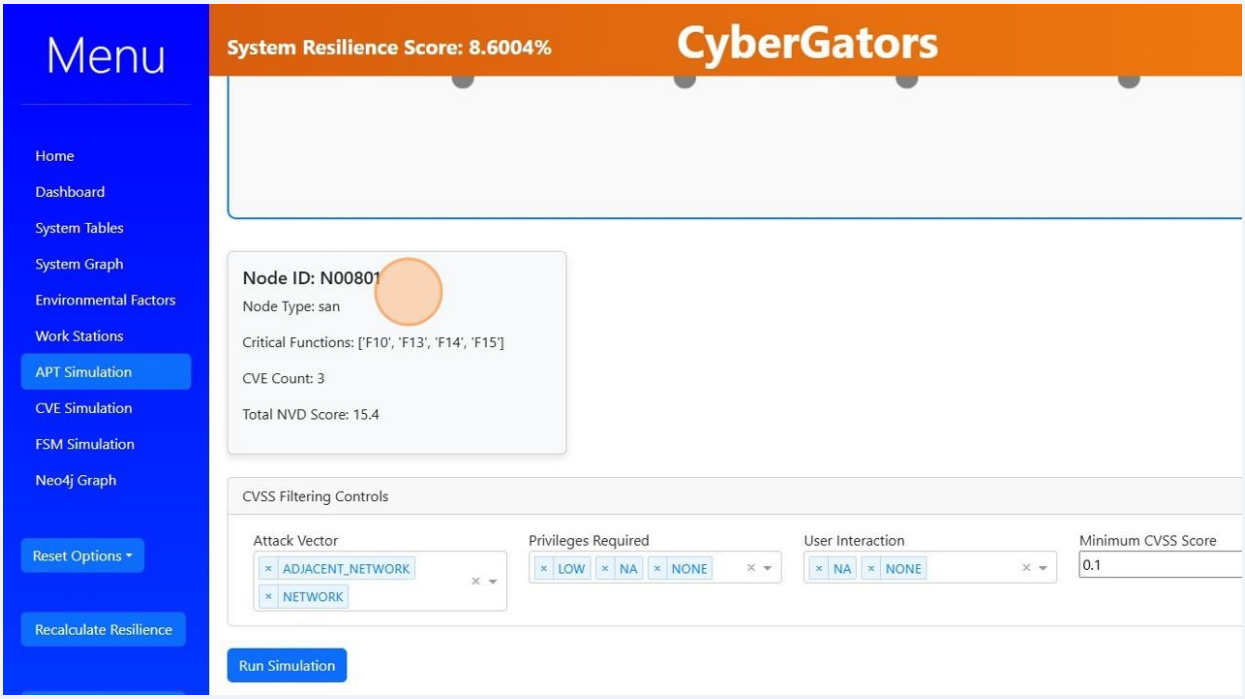
23 Click "Select CVE..." to select which CVE to exploit within the system.



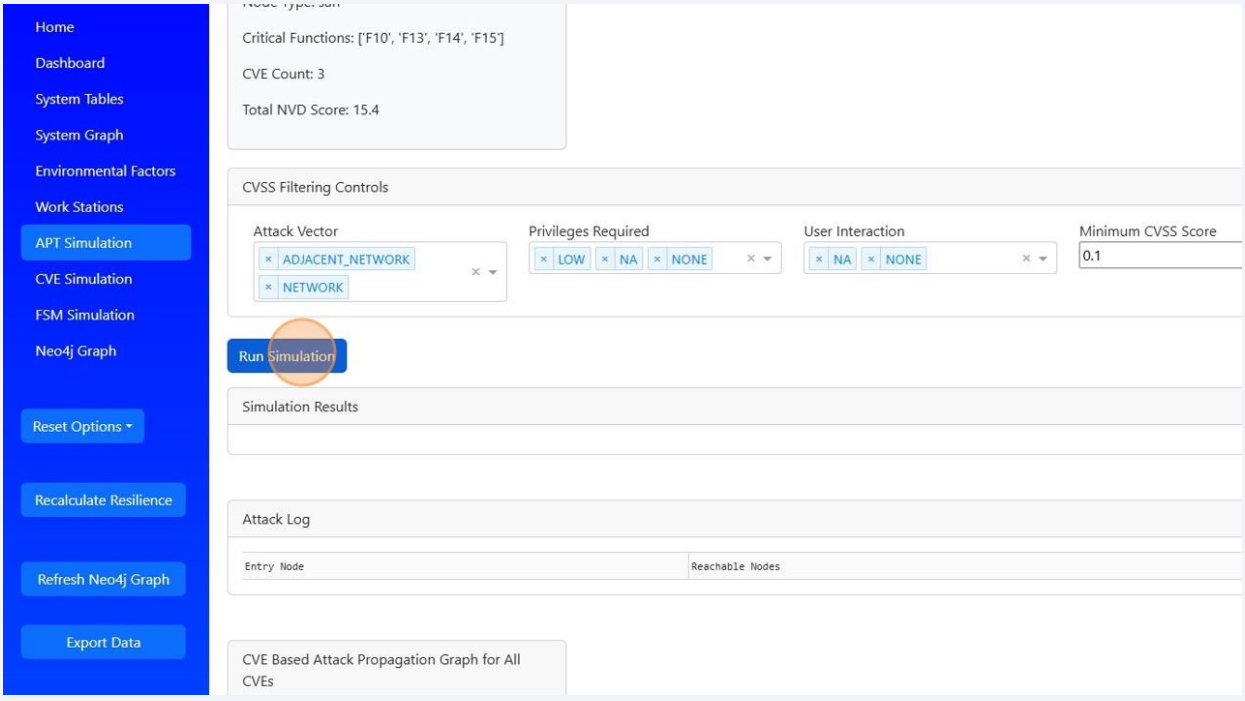
24 View the nodes that contain that exploited CVE within the system on the first graph



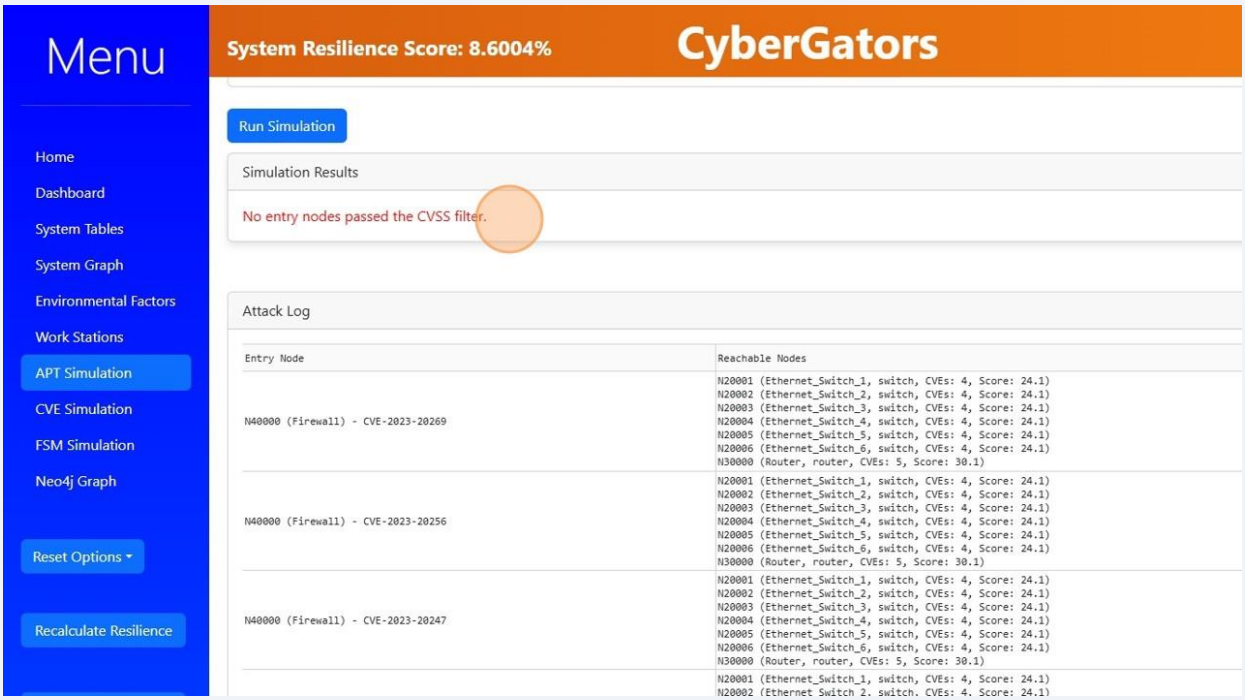
25 Selecting a highlighted node will reveal more information about the node that is under attack, as well as information about the associated attack vector, required privileges, if user interaction is necessary for exploitation, and minimum CVSS score.



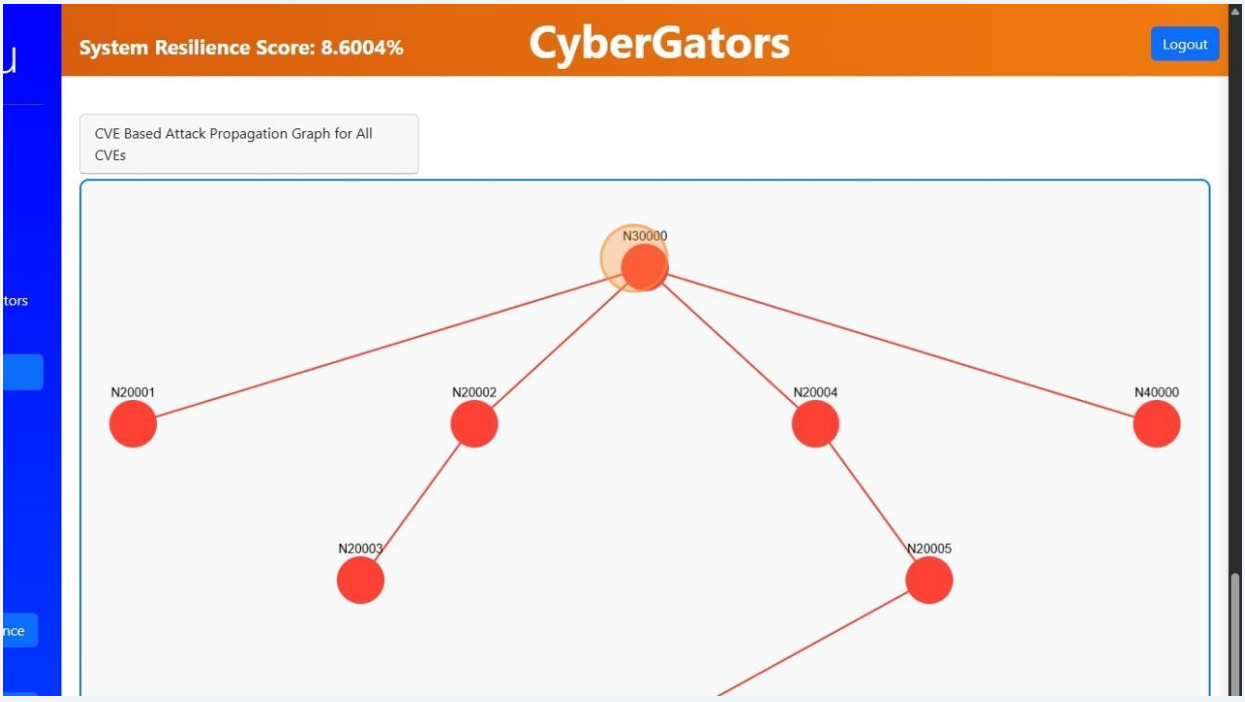
26 To run the exploitation simulation on the selected node, click "Run Simulation"



27 The simulation results will populate below, along with a comprehensive attack log and the nodes/CVEs attacked and exploited

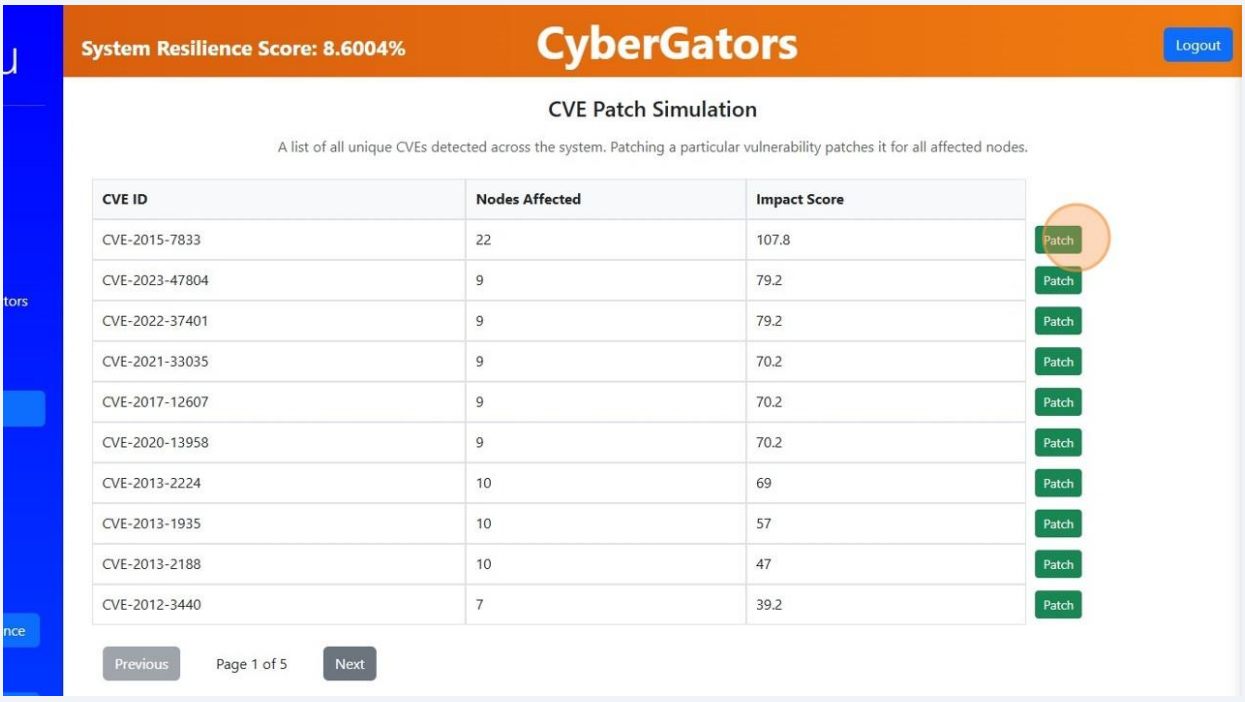


28 Scroll down further to see the CVE Based Attack Propagation Graph for All CVEs. This shows the traversal through the system based on the node IDs and the exploited CVE and attack log.

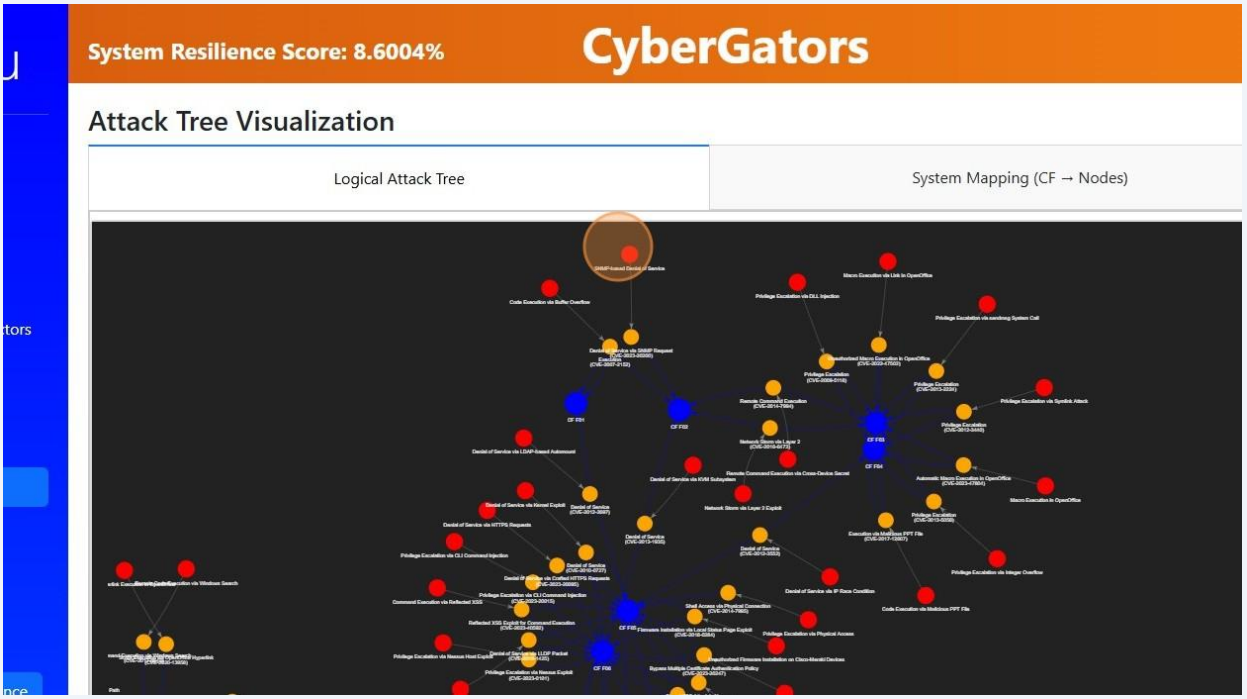


29 At the very bottom, when a node is selected, more information about the node populates, including the node type, ID, total CVEs, total CVE score, any critical functions, if it exists on a server rack, or requires a certain privilege level to access.

30 Next, to view the CVE Patch Simulation, click "CVE Simulation." To simulate patching a CVE within the system, click "Patch" next to the CVE you wish to patch.

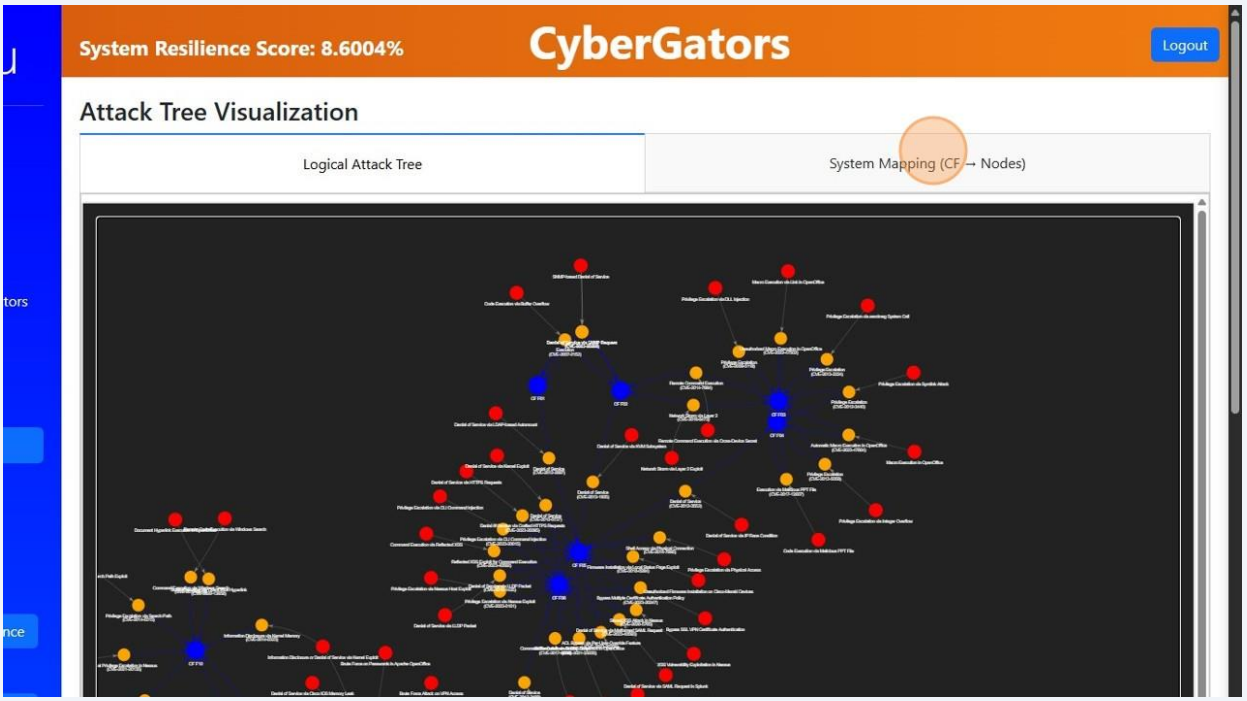


31 Next, to view the Free State Machine Simulation and Attack Tree Visualization, click "FSM Simulation" from the sidebar. Select a starting node on the edge of the graph to begin traversing the Attack Tree (see Logical Attack Tree)

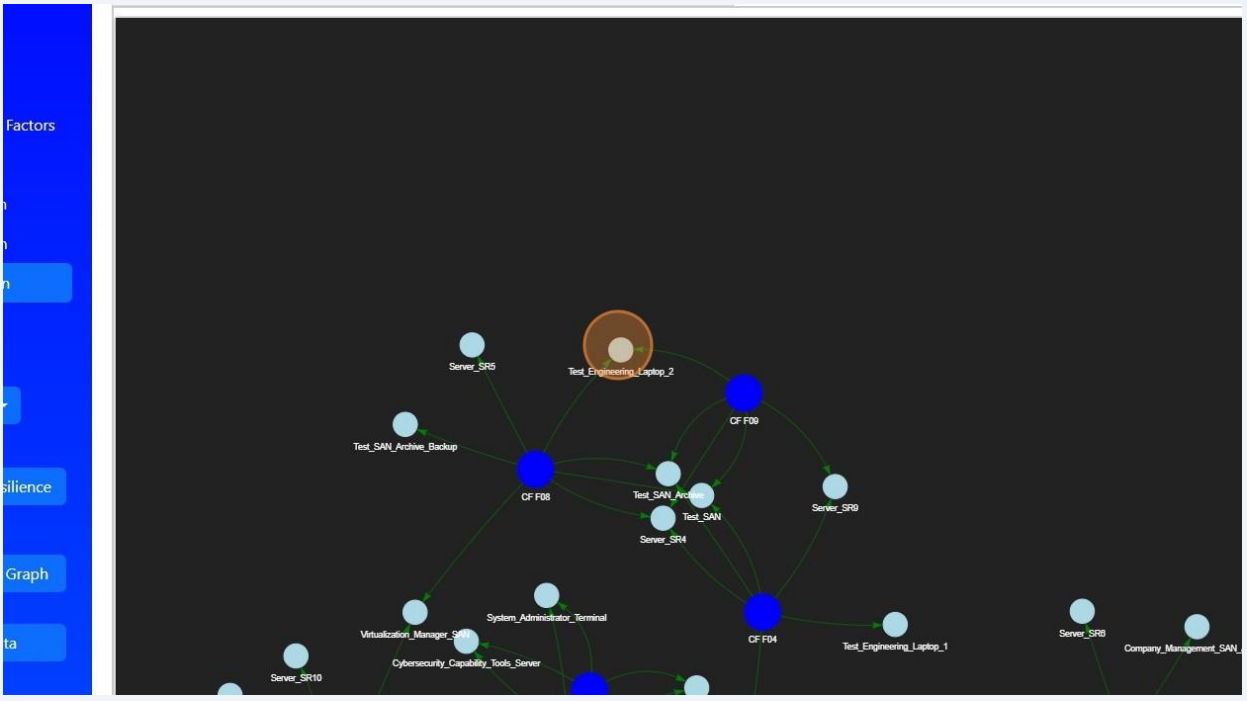


32 Selecting the next node reveals the traversal information, including the penetration tactic, tactic, exploited CVE, and probability of success

33 To view the association between the nodes and Critical Functions, click "System Mapping (CF → Nodes)"



34 To start the simulation, select the starting node.



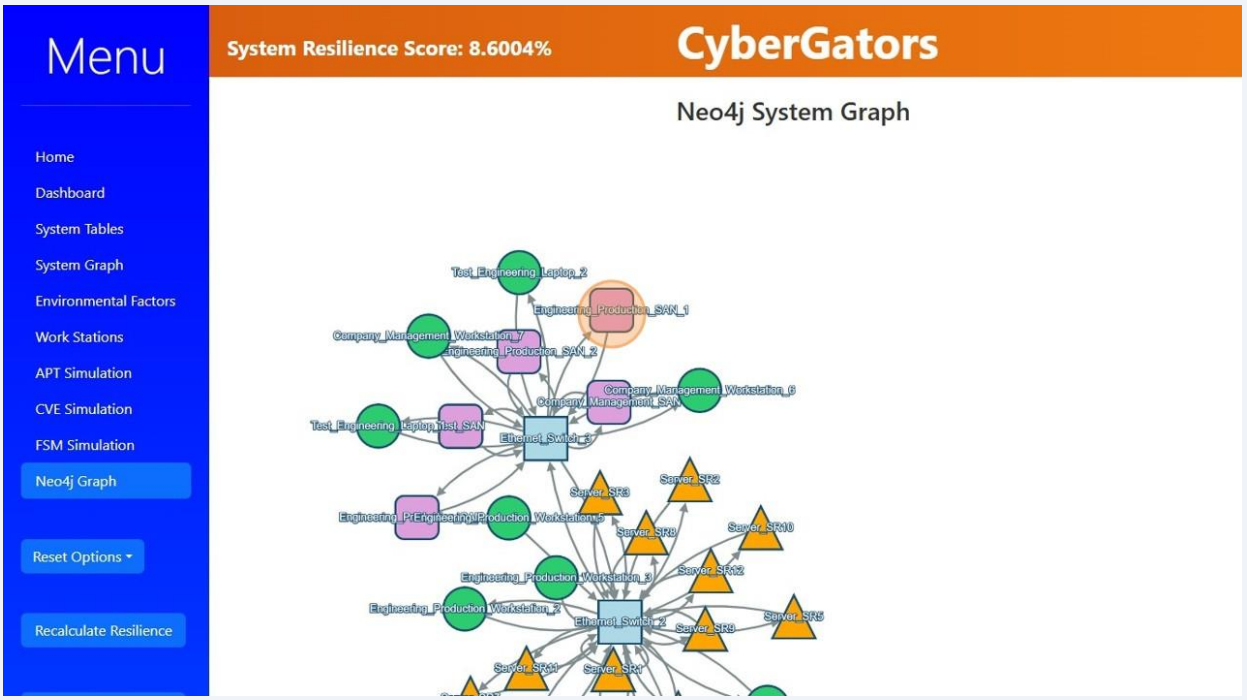
35 Select the traversal node

36 This reveals more information about the traversal nodes and total CVE score on a given node, as well as directed edges along with the attack could traverse.

Neo4j View

37 To view the Neo4j graph which is linked to the database containing the system, click "Neo4j Graph" on the sidebar.

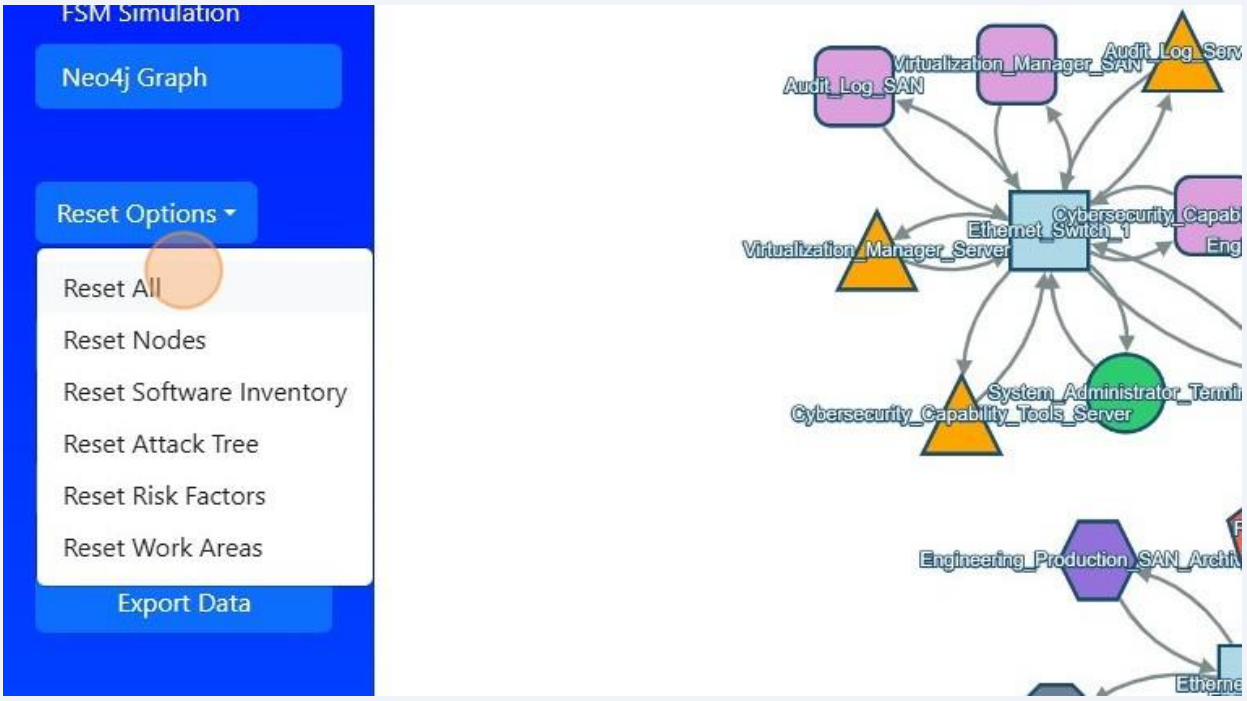
“The Neo4j graph is a prototype view combining system components through a database-backed interface. It currently displays system nodes and directed relationships but is still under active development.”



Reset, Export, and Logout

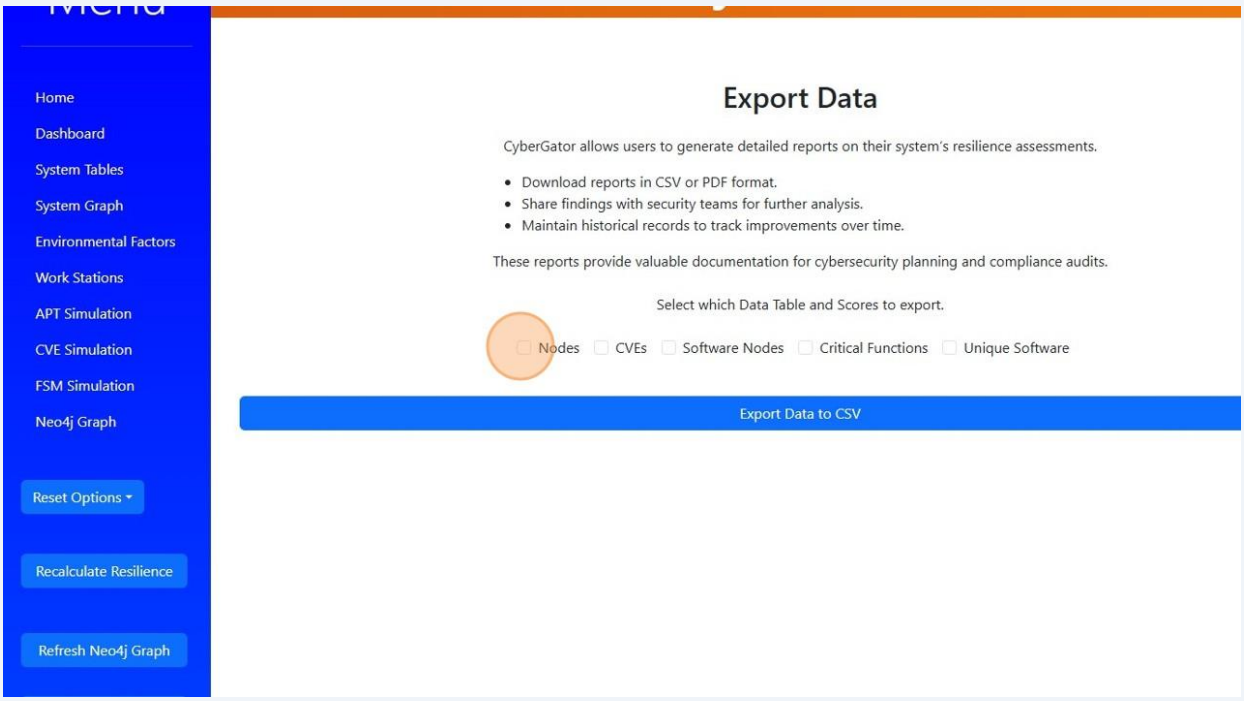
38 To reset any of the changes made back to the original system settings, click "Reset Options" in the sidebar.

To reset any of the changes made back to the original system settings, click "Reset Options" and select which components to reset.



39 To recalculate the resilience of the system, click "Recalculate Resilience" on the sidebar, and notice that the System Resilience Score updates on the top bar.

40 To export data, select the "Export Data" button the sidebar, select the data tables to export, and then Export to CSV button. To capture the System Resilience Score, navigate back to the Dashboard and export the pie chart as described earlier.



41 To log out, simply click "Logout" in the upper righthand corner.

FAQs

1. **I added CVEs but don’t see a change in the resilience score. Why?**

After making changes to nodes or CVEs, you must click “Recalculate Resilience” to update scores. Without that step, the simulation results won't refresh.

2. **How do I undo a change or start over?**

Use the “Reset System” button on any page to restore the original data. This reverts to backup JSON files and clears simulation state.

3. **What do the resilience scores actually mean?**

CyberGator uses a combination of fuzzy logic, Bayesian inference, and finite state machines to estimate system resilience. These scores are experimental and designed for educational and research purposes—not for production decision-making.

4. **Can I simulate multiple attacks or changes at once?**

Yes. You can inject multiple CVEs, change critical functions, and modify environmental risk factors before recalculating the score. All changes are reflected in the next simulation cycle.

5. **Is the Neo4j database live?**

Not yet. The Neo4j view is a prototype meant to illustrate future integration. The current system runs off structured JSON and CSV files.

6. **Where are my changes stored?**

Changes are written to output JSON files located in data/json/output, which persist across sessions until reset. These files include node metrics, system scores, and resilience breakdowns.

7. **Can I export my results?**

Yes. Navigate to the Export tab to download tables in CSV format, including nodes, scores, and CVE data.

8. **What if I want to test my own system?**

This version of CyberGator is a prototype based on a single System Under Evaluation (SUE) from the CRAM Challenge. Future versions will include the ability to upload and simulate your own system architecture. Users can also manipulate the uploaded system so that it accurately reflects their own enterprise system.

Transition Plan/Next Steps

Following the completion of the CyberGator project, primary ownership will transition to Ozlem Polat, who plans to continue developing the application as part of her doctoral research with the U.S. Army Research Lab. Her focus will include extending the Neo4j database integration to support attack graph traversal and advanced CVE exploitation modeling—two components originally scoped but deferred during the initial project phase.

A complete README is included in the repository. It contains installation instructions, usage guidelines, a project overview, and frequently asked questions to assist future contributors or researchers. While no formal onboarding process was conducted, the documentation and

modular design of the codebase support continued work without requiring in-person knowledge transfer.

At this time, there is no broader development roadmap beyond Ozlem’s research. Other project team members will not be continuing development post-submission. The current application is stable and functional, with clearly documented backlog items for future enhancements.

The final deliverable will be showcased in a public project presentation and university-hosted demo event, with attendance from faculty advisors and course instructors. This event will mark the formal close of the undergraduate development effort and serve as a final checkpoint for demonstrating the tool's core capabilities and future research potential.

Acknowledgements

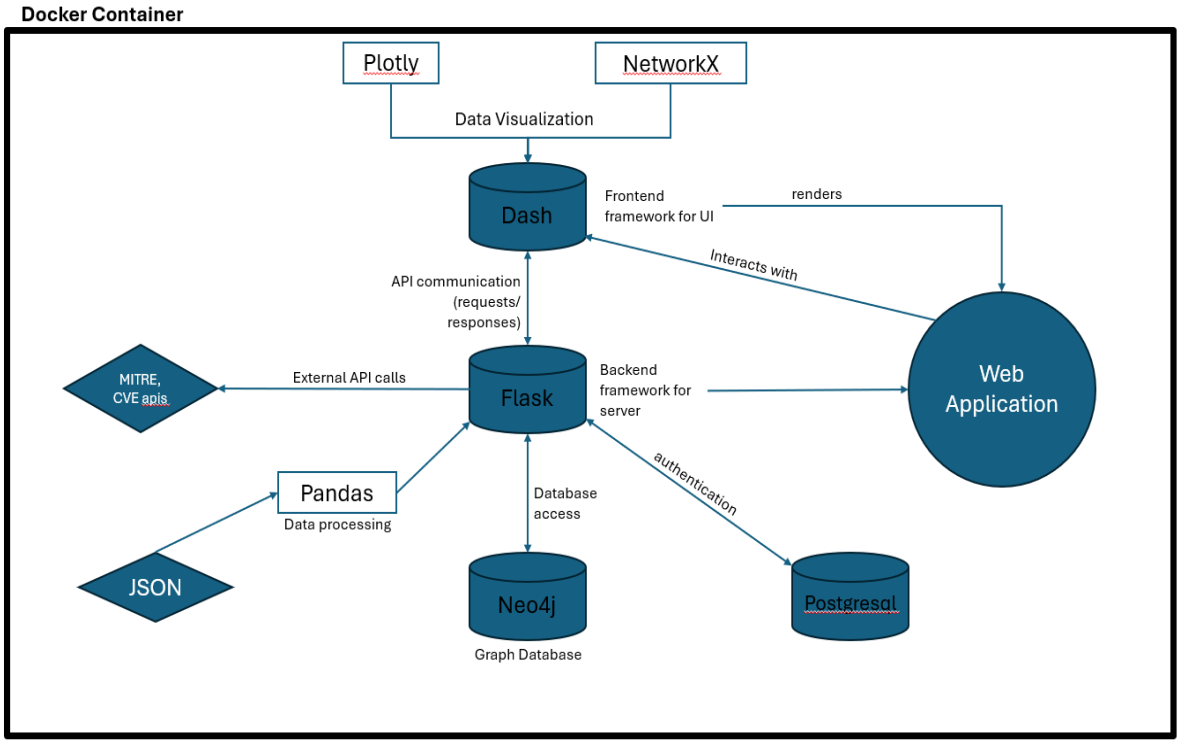
We would like to thank Dr. Unger, who worked closely with Jessica and Ozlem as they continued their independent research into system resilience and security. His guidance helped us reframe our work as a formal computer science problem, pushing us to approach the challenge through the lens of algorithms, problem decomposition, and computational modeling. He provided critical insight on problem formulation, helped us identify suitable algorithmic techniques, and encouraged us to think rigorously about the structure and complexity of our data. His mentorship significantly shaped the research direction of our project, and we are grateful for the time and attention he dedicated to supporting our progress. Thank you, Dr. Unger!

We would primarily like to thank Dr. Cheryl Resch for her unwavering support of our research and work over the last two semesters. She facilitated our travels to NAVSEA Dahlgren last

semester and agreed to oversee our Senior Design project and support the development of our complex algorithms, on top of her busy schedule as an Associate Professor. She provided us with support and advice consistently over the last eight months as we tackled hurricanes and the complexity of senior design and contributed her feedback through the lens of her professional experiences as a security expert and academic prowess as a university professor. Thank you very much, Dr. Resch! It was a privilege to work with you so closely.

Appendices

System Architecture Final Diagram



Sample Code Fuzzy Set

```
{
    "flood_risk": {
        "Yes": { "left": 0.8, "peak": 1.0, "right": 1.0 },
        "No": { "left": 0.0, "peak": 0.0, "right": 0.2 },
        "NA": {"left": 0.0, "peak": 0.0, "right": 0.0}
    },
    "unlocked_doors": {
        "Yes": { "left": 0.4, "peak": 0.5, "right": 0.6 },
        "No": { "left": 0.0, "peak": 0.0, "right": 0.2 },
        "NA": {"left": 0.0, "peak": 0.0, "right": 0.0}
    },
    "security_guard_present": {
        "Part-time": { "left": 0.3, "peak": 0.5, "right": 0.7 },
        "Full-time": { "left": 0.0, "peak": 0.0, "right": 0.2 },
        "No": { "left": 0.7, "peak": 1.0, "right": 1.0 },
        "NA": {"left": 0.0, "peak": 0.0, "right": 0.0}
    },
    "alarm_coverage": {
        "None": { "left": 0.6, "peak": 0.8, "right": 1.0 },
        "Limited": { "left": 0.3, "peak": 0.5, "right": 0.7 },
        "Full": { "left": 0.0, "peak": 0.1, "right": 0.2 },
        "NA": {"left": 0.0, "peak": 0.0, "right": 0.0}
    },
    "access_control": {
        "Low": { "left": 1.2, "peak": 1.5, "right": 1.8 },
        "Medium": { "left": 0.3, "peak": 0.5, "right": 0.7 },
        "High": { "left": 0.0, "peak": 0.1, "right": 0.2 },
        "NA": {"left": 0.0, "peak": 0.0, "right": 0.0}
    },
    "key_lock_usage": {
```

Resilience Calculation Steps from resilience_calcluator.py

```
# ------------------------------------------------------------
# Overview:
# This code calculates resilience scores for nodes in a network based on multiple metrics:
# 1. Fuzzy Logic Calculation: For environmental risk factors.
# 2. Vulnerability Scores (CVE): Based on NVD CVE scores.
# 3. Centrality: How important a node is in the network.
# 4. Connectedness: How connected the node is to others.
# 5. Criticality: Importance based on associated critical functions.
# 6. Switch Dependency: Whether the node depends on switches for access.
# 7. Redundancy: Whether the node has redundancy built in (e.g., SAN backup).
# The final resilience score combines all these factors.
# See below the code for the step-by-step explanation of each metric.
# ------------------------------------------------------------

# Input files
# input_file = os.path.join("src", "maindata", "Risk_Factors.json")
# fuzzy_file = os.path.join("src", "maindata", "Fuzzy_Set.json")
# nodes_file = os.path.join("src", "maindata", "Nodes_Complete.json")

# Load nodes data
# with open(nodes_file, 'r') as file:
#     nodes_data = json.load(file)
```

Calculate Environmental Risk with Fuzzy Logic

```python
def calculate_environmental_risk_with_fuzzy_logic(work_area, fuzzy_sets):
    """
    Calculate the environmental risk for a given work area using fuzzy logic.
    """
    risk_factors = work_area['Risk_Factors']
    total_risk_score = 0
    for factor, value in risk_factors.items():
        if factor in fuzzy_sets and value in fuzzy_sets[factor]:
            fuzzy_set = fuzzy_sets[factor][value]
            membership_value = fuzzy_set['peak']
            total_risk_score += membership_value
    return total_risk_score

def calculate_all_work_areas_risk_fuzzy(input_file, fuzzy_file):
    """
    Calculate the environmental risk for all work areas using fuzzy logic.
    """
    with open(input_file, 'r') as file:
        work_area_data = json.load(file)

    with open(fuzzy_file, 'r') as fuzzy_file:
        fuzzy_sets = json.load(fuzzy_file)

    work_areas = work_area_data['work_areas']
    environmental_risk_scores_fuzzy = {}

    for area in work_areas:
        work_area_name = area['Work_Area']
        environmental_risk = calculate_environmental_risk_with_fuzzy_logic(area, fuzzy_sets)
        environmental_risk_scores_fuzzy[work_area_name] = round(environmental_risk, 1)

    return environmental_risk_scores_fuzzy
```

Calculate Resilience Scores

```python
# Step 10: Final resilience score calculation
def calculate_resilience_scores(nodes_data, graph, fuzzy_scores, critical_function_weights):
    """
    Calculates resilience scores for all nodes by combining the fuzzy environmental risk score,
    CVE vulnerability, centrality, connectedness, switch dependency, and redundancy.

    Calculate resilience scores for all nodes by reducing a starting score of 100
    based on penalties for CVE scores, centrality, connectedness, etc., and then
    applying environmental risk penalties at the end.
    """
    # Start with a base score of 100 for each node
    base_score = 100

    # Pre-calculate graph metrics
    centrality_scores = calculate_centrality_scores(graph)
    connectedness_scores = calculate_connectedness_scores(graph)

    resilience_scores = []

    for node in nodes_data:
        node_id = node['node_id']
        resilience_score = base_score

        # Apply penalties (deduct from 100)
        cve_score = calculate_vulnerability_score(node)
        centrality = centrality_scores.get(node_id, 0)
        connectedness = connectedness_scores.get(node_id, 0)
        switch_dep = calculate_switch_dependency(node)
        redundancy = calculate_redundancy(node)
        criticality = calculate_criticality(node, critical_function_weights)
```

```python
        # Deduct points for each factor
        resilience_score -= cve_score * 0.4  # CVE score penalty (weighted)
        resilience_score -= centrality * 0.2  # Centrality penalty (weighted)
        resilience_score -= connectedness * 0.2  # Connectedness penalty (weighted)
        resilience_score -= switch_dep * 0.2  # Switch dependency penalty
        resilience_score -= criticality * 0.3  # Criticality penalty
        resilience_score += redundancy * 0.2  # Redundancy reduces penalty

        # Calculate environmental risk for this node
        environmental_risk = calculate_node_environmental_risk(node, fuzzy_scores, critical_function_to_work_area)

        # Final adjustment: Divide resilience score by environmental risk (1 + risk factor)
        resilience_score /= (1 + environmental_risk)

        # check to make sure the resilience score doesn't drop below 0
        resilience_score = max(resilience_score, 0.001)

        # Store the calculated resilience score
        resilience_scores.append({
            'node_id': node_id,
            'node_name': node['node_name'],
            'resilience_score': round(resilience_score, 5),
        })

    return resilience_scores
```

Calculate Resilience

```python
# Step 1: Function to calculate system resilience score
def calculate_system_resilience(resilience_scores):
    """
    Calculate the overall system resilience score by summing up all node resilience scores.
    """
    total_resilience = sum(node['resilience_score'] for node in resilience_scores)

        # Normalization: Divide the total resilience by the total number of nodes
    total_number_of_nodes = len(resilience_scores)

    # Normalize the total system resilience score
    normalized_system_resilience = total_resilience / total_number_of_nodes

    return round(normalized_system_resilience, 5)
```

Biographies

Ozlem Polat: Ozlem is a senior Computer Science student with a concentration in Data Science. She brings a decade of experience in regulated industries including food, pharmaceuticals, and alcoholic beverages, where she focused onIT compliance and data security. She has worked with Python, MATLAB, R, Jupyter Notebook, and React, to name a few, and contributed to CyberGator by acting as the project lead and SCRUM Master. She also supported database development, research, and integration. Currently a data analyst intern, Ozlem, hopes to apply data science to complex systems and she plans to build on this project as part of her PhD thesis. Her long-term goals include research and advanced study in data science. After graduation, Ozlem will continue her research in cyber resilience at the University of Florida in partnership with the Army Research Laboratory.

Jessica Lourenço: Jess is a senior Computer Science student and a software engineer with a strong background in backend development, cloud infrastructure, and resilient system design. For CyberGator, she led the backend architecture and infrastructure, implementing core algorithms and modeling the data layer. Her contributions were closely tied to independent research in cyber resilience, with a focus on formal methods, attack graph modeling, and algorithm design. Outside of the project, Jess works as a Java software engineer at Dock, contributes to open-source tooling, and volunteers for Lacrei Saúde—an inclusive health platform for LGBTQIA+ communities in Brazil. With prior experience as a product manager, Jess brings a user-centered mindset to all of her technical work and plans to continue growing as a backend engineer after graduation.

Samson Carter: Samson is a senior Computer Science student at the University of Florida with prior experience in product management and machine learning. He contributed to CyberGator as the project manager and led development of the GUI prototype in Figma. His past work includes optimizing machine learning models, such as Random Forest, XGBoost, and artificial neural networks, for predicting housing prices, with a focus on feature engineering and ensemble methods. Samson has experience coding in C++, JavaScript, and Python, and plans to pursue a career in product management, ideally within the defense sector. Outside of academics, he is passionate about music, creative writing, and community engagement.

Shayan Akhoondan: Shayan is a Computer Science student with experience in C++, Python, and TensorFlow. For the CyberGator project, he contributed heavily to UI design and simulation research. He previously developed a convolutional neural network to analyze MRI brain scans, achieving high classification accuracy. His technical strengths include model evaluation, precision analysis, and using GPU-accelerated training environments. Shayan plans to pursue graduate study in AI or cybersecurity and values hands-on work that supports national defense missions. He views the CRAM project as a chance to apply theoretical knowledge to high-impact systems and develop tools with real-world utility.

Andrew Ballard: Andrew is a Computer Science student with experience in front-end development, UI/UX design, and machine learning. On CyberGator, he was responsible for building interactive components in Dash and integrating front-end features with the data model. His previous work includes console-based games and neural network development using TensorFlow to classify medical imaging data. He also served as SCRUM master on a separate team project, where he led meetings and managed technical progress. Andrew hopes to enter the

cybersecurity field after graduation and sees this project as a pivotal learning experience that bridges academic theory with real-world problem-solving.