Project Title: Attack Graph Augmented Vulnerability Prioritization Engine

---

Core Idea:

- **Link vulnerabilities into attack paths** (attack graphs) to find vulnerabilities that are **critical to attacker success**, not just high CVSS scores.

Component	Role
Vulnerability Ingestion Pipeline	Collects vulnerability scans (Qualys, Nessus, Burp Suite, etc)
Network/Asset Graph Builder	Builds system graphs (devices, applications, communications, etc)
Attack Path Generator	Builds possible chains of exploits between assets
Graph Risk Analyzer	Analyzes which vulnerabilities enable critical attack paths
Prioritized Remediation Engine	Prioritizes patches that break dangerous paths

---

Component Details:

1. **Vulnerability Ingestion Pipeline**:
    - Normalizes scan data:
        - Maps vulnerabilities to assets.
        - Captures versions, exposure levels, etc.
2. **Network/Asset Graph Builder**:
    - Builds a directed graph:
        - Nodes = assets.
        - Edges = communication links or trust relationships.
3. **Attack Path Generator**:
    - Builds potential kill chains:
        - Initial foothold → privilege escalation → lateral movement → crown jewels.
4. **Graph Risk Analyzer**:
    - Identifies:
        - Choke points (assets whose compromise leads to cascade failures).
        - Critical paths requiring minimal attacker effort.
5. **Prioritized Remediation Engine**:
    - Breaks key attack paths first:
        - Focus patches on maximum impact for defense.

Overall System Flow:

- Input: Vulnerabilities + asset topology
- Output: Patch priorities based on attacker pathways
- Focus: **Break attack chains, not just patch based on CVSS**

Internal Functioning of Each Module:

1. Vulnerability Ingestion Pipeline

- **Data gathering**:
  - Integrations with:
    - Nessus
    - OpenVAS
    - Qualys
    - Etc
  - Normalize findings into internal schema:
    - e.g., Asset-Vulnerability pairs

---

2. Network/Asset Graph Builder

- **Graph Structure**:
  - Nodes:
    - Systems
    - Services
    - Accounts
    - Etc
  - Edges:
    - Network links
    - Authentication/authorization paths
    - Etc

---

3. Attack Path Generator

- **Path Construction**:
  - Based on real attacker TTPs (e.g., MITRE ATT&CK).
  - Examples:
    - Phish user ➔ Use password reuse ➔ Access VPN ➔ RCE on app server.
- **Algorithms**:
  - Shortest attack paths.
  - Maximum risk paths (minimal effort, maximum payoff).
  - Etc.

---

4. Graph Risk Analyzer

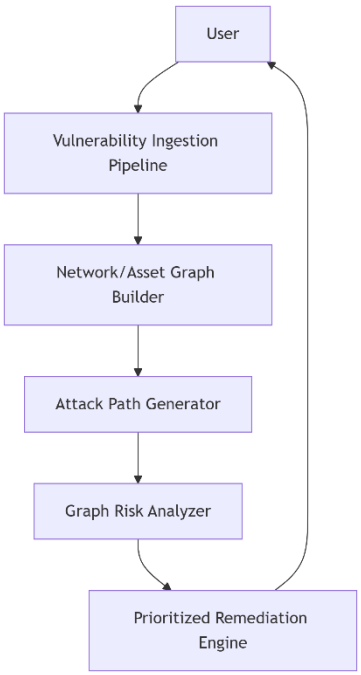
- **Prioritization logic**:
  - High priority to vulnerabilities that:
    - Shorten attacker paths dramatically.
    - Provide escalations (local → admin).

---

5. Prioritized Remediation Engine

- **Patch Planning**:
  - Select patches breaking entire attack chains first.

Component Diagram

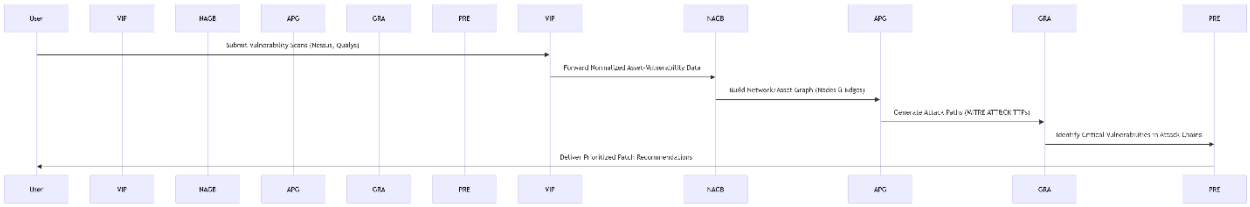


- **Flow Top to Bottom:**
  1. **User** submits vulnerability scans.
  2. **Vulnerability Ingestion Pipeline** normalizes data (Nessus, Qualys, etc.).
  3. **Network/Asset Graph Builder** constructs asset topology (nodes, edges).
  4. **Attack Path Generator** simulates exploit chains (MITRE ATT&CK-based paths).
  5. **Graph Risk Analyzer** identifies critical vulnerabilities (choke points, privilege escalations).

6. **Prioritized Remediation Engine** generates actionable patch recommendations.
7. **User** receives prioritized remediation list.

- **Key Feedback Loop:**
  - **Remediation Engine** → **User**: After patching, new scans restart the cycle, refining attack graphs iteratively.

Sequence Diagram



1. **User** submits vulnerability scan data to the **Vulnerability Ingestion Pipeline**.
2. **Pipeline** normalizes the data and sends it to the **Network/Asset Graph Builder**.
3. **Graph Builder** constructs a network topology (e.g., servers, services, trust relationships, etc) and forwards it to the **Attack Path Generator**.
4. **Attack Path Generator** creates potential exploit chains (e.g., phishing → VPN compromise → RCE on a critical server).
5. **Graph Risk Analyzer** identifies vulnerabilities that enable the most dangerous paths and sends them to the **Prioritized Remediation Engine**.
6. **Remediation Engine** generates a prioritized patch list (e.g., "Patch Server X first to break lateral movement") for the **User**.

Detailed Project Description: Attack Graph Augmented Vulnerability Prioritization Engine

A system that prioritizes vulnerabilities based on their role in enabling attack paths. By modeling attack graphs and identifying critical choke points, this system enables organizations to focus remediation efforts on vulnerabilities that pose the highest strategic risk.

---

1. System Architecture Overview

Core Components & Interactions

1. **Vulnerability Ingestion Pipeline**
    - *Inputs*: Vulnerability scans (Nessus, Qualys, OpenVAS, Burp Suite, etc).
    - *Outputs*: Normalized asset-vulnerability pairs.
2. **Network/Asset Graph Builder**
    - *Inputs*: Asset metadata, network configurations.
    - *Outputs*: Directed graph of assets, services, and trust relationships.
3. **Attack Path Generator**
    - *Inputs*: Graph topology, MITRE ATT&CK TTPs.
    - *Outputs*: Potential attack paths (e.g., phishing → VPN compromise → lateral movement).
4. **Graph Risk Analyzer**
    - *Inputs*: Attack paths, vulnerability data, etc.
    - *Outputs*: Prioritized vulnerabilities (choke points, privilege escalation enablers, etc).
5. **Prioritized Remediation Engine**
    - *Inputs*: Critical vulnerabilities.
    - *Outputs*: Actionable patch recommendations.

Integration Flow:

1. Scans → Ingestion Pipeline → Graph Builder → Attack Path Generator → Risk Analyzer → Remediation Engine.
2. Feedback loop: Post-remediation scans refine attack graphs iteratively.

---

2. Component Implementation Details

2.1 Vulnerability Ingestion Pipeline

Objective: Aggregate and normalize vulnerability data from diverse sources.

Tools & Workflow:

- **Data Sources**:
  - *Tools*: Nessus API, Qualys Cloud Platform, etc.
  - *Normalization*: Map findings to a common schema (e.g., CVE ID, asset IP, CVSS score, etc).
- **Example Code**:

```
def normalize_nessus_scan(scan_data):
    normalized = []
    for finding in scan_data["vulnerabilities"]:
        normalized.append({
            "cve": finding["cve_id"],
            "asset_ip": finding["asset"]["ip"],
            "severity": finding["severity"],
            "description": finding["description"]
        })
    return normalized
```

- **Validation**: Ensure all assets and vulnerabilities are uniquely mapped.

2.2 Network/Asset Graph Builder

Objective: Model assets and interactions as a directed graph.

Implementation Steps:

1. **Graph Structure**:
    - *Nodes*: Assets (servers, workstations, etc), services (Apache, MySQL, etc), user accounts, etc.
    - *Edges*: Network connections (SSH, RDP, etc), trust relationships (domain admin privileges, etc).
2. **Data Sources**:
    - *Network Scans*: Nmap for topology discovery.
    - *Active Directory*: LDAP queries for user-role mappings.
3. **Graph Database**:
    - Use Neo4j or Amazon Neptune for storage and querying.
    - *Example Query [cypher]*:
      ```
      MATCH (n:Server)-[r:TRUSTS]->(m:DomainController) RETURN n, r, m
      ```

2.3 Attack Path Generator

Objective: Simulate attacker kill chains using MITRE ATT&CK tactics.

Implementation Steps:

1. **Path Construction**:
    - *Algorithms*: Breadth-First Search (BFS) or Dijkstra's algorithm to find shortest paths.
    - *TTP Integration*: Map vulnerabilities to MITRE ATT&CK techniques (e.g., CVE-2023-XXXX → T1190: Exploit Public-Facing Application).
2. **Simulation Tools**:
    - Use BloodHound for Active Directory attack path analysis.
    - *Example Code*:
      ```
      def generate_attack_paths(graph):
          paths = nx.all_simple_paths(graph, source="Phished_Workstation",
      target="Domain_Admin")
          return list(paths)
      ```

2.4 Graph Risk Analyzer

Objective: Identify vulnerabilities critical to high-risk attack paths.

Implementation Strategies:

1. **Centrality Analysis**:
    - Use betweenness centrality to find choke points (nodes in most shortest paths).
2. **Privilege Escalation Detection**:
    - Flag vulnerabilities enabling admin access (e.g., CVE-2023-YYYY: Local Privilege Escalation).
3. **Example Code**:

```
import networkx as nx
G = nx.read_graphml("attack_graph.graphml")
centrality = nx.betweenness_centrality(G)
critical_nodes = [node for node, score in centrality.items() if score > 0.
1]
```

2.5 Prioritized Remediation Engine

Objective: Recommend patches that disrupt critical attack paths.

Implementation Steps:

1. **Impact Scoring**:
    - Rank vulnerabilities by the number of attack paths they disrupt.
    - *Formula*: `Priority Score = (Paths Disrupted × 0.6) + (CVSS Score × 0.4)`
2. **Integration with Ticketing Systems (optional)**:
    - Auto-generate Jira/ServiceNow tickets with remediation steps.
    - *Example Output [markdown]*:
      `**Priority 1**: Patch CVE-2023-XXXX on Server 10.0.0.5`

```
- **Impact**: Breaks 12 attack paths, including lateral movement to
domain controllers.
- **Remediation**: Apply Microsoft KB123456.
```

---

3. Evaluation Metrics

1. **Attack Path Reduction**:
    - Percentage reduction in viable attack paths post-remediation.
2. **Remediation Efficiency**:
    - Time-to-patch critical vulnerabilities vs. traditional CVSS-based prioritization.
3. **False Positives**:
    - Number of non-critical vulnerabilities flagged as high priority.

---

4. Challenges & Mitigation (optional)

- **Scalability**:
    - Use distributed graph processing (Apache Giraph) for large networks.
- **Data Freshness**:
    - Schedule nightly network scans and vulnerability updates.
- **Complexity**:
    - Simplify outputs with visual dashboards highlighting top 5 critical vulnerabilities.

---

5. Tools & Technologies (e.g.)

- **Graph Database**: Neo4j, Amazon Neptune, etc.
- **Path Analysis**: BloodHound, NetworkX, etc.
- **APIs**: FastAPI, Nessus API, etc.
- **Visualization**: Grafana, Elastic Kibana, etc.