Security Vulnerability Assessment Tool for Cloud Services

Huilin Chang
MS in Cybersecurity, College of Computing, Georgia Institution of Technology
hchang308@gatech.edu

Abstract

With growing reliance on cloud platforms like AWS, businesses are faced with security vulnerabilities predominantly caused by misconfigurations. Current existing assessment tools lack comprehensive real-time monitoring and user-specific customization. This paper presents the development and evaluation of a Security Vulnerability Assessment Tool tailored for AWS environments. Through event-driven, real-time monitoring and integration with other AWS services, this tool aims to not only detect but effectively mitigate security vulnerabilities. Preliminary evaluations, presented in two case studies, demonstrate this tool's accuracy in identifying misconfigurations.

Keywords— *Cloud service, Security, AWS, Tool*

I. INTRODUCTION

Use of cloud services has surged in the last decade, predominantly with Amazon Web Services (AWS), Azure, and Google Cloud Platform (GCP). This has caused a paradigm shift in how businesses operate, with companies attracted to the flexibility and scalability of these platforms. However, with greater widespread adoption comes various potential security vulnerabilities, primarily through misconfigurations.

A Review of Recent Cloud Security Breaches

The landscape of cloud security is marked by a series of notable breaches, emphasizing the need for robust security measures. A chronological examination of these incidents offers insight into the nature and challenge of cloud vulnerabilities [1-7]:

- June 2022: A former AWS employee's involvement in the Capital One breach highlighted the risks of insider threats.
- May 2022: The Pegasus Airlines breach exposed 23 million files, demonstrating the scale of data at risk.
- December 2021: FlexBooker's breach impacted 3 million users, showcasing the extensive reach of cloud vulnerabilities.
- August 2021: SeniorAdvisor's exposure of personal data affected over 3 million senior citizens, a vulnerable group.
- July 2021: PeopleGIS exposed sensitive information from over 80 municipalities, underlining the threats to public sector data.
- June 2021: The data breaches of Turkish cosmetics retailer Cosmolog Kozmetik and COVID testing sites emphasized the diversity of personal health data at risk.
- February 2021: LogicGate's breach and data exposure by Prestige Software in November 2020 further stressed the urgency for improved cloud security.

These incidents, often resulting from simple misconfigurations and oversight, underline the far-reaching consequences of security lapses in cloud environments.

Limitations of Current Cloud Security Tools

While existing tools for cloud security are functional, they often lack the sophistication needed for comprehensive, real-time monitoring and fail to offer necessary customization for varying user environments. This paper focuses on addressing these gaps by introducing a Security Vulnerability Assessment Tool tailored for AWS. This tool aims to not only identify but also effectively mitigate security vulnerabilities by integrating event-driven real-time monitoring with AWS services. Preliminary evaluations, demonstrated through two case studies, indicate significant progress in detecting misconfigurations.

Context and the Need for Continuous Innovation in Cloud Security

As the cloud computing landscape evolves rapidly, security remains a paramount concern for organizations utilizing cloud services. Tools like AWS Self-Service Security Assessment Solutions v2.0 and AWS Inspector, despite their capabilities, exhibit limitations. For example, AWS Self-Service Security Assessment Solutions v2.0 lacks comprehensive service coverage, customization, real-time monitoring, and has a potentially unintuitive user interface for novices. AWS Inspector, initially focused on Amazon EC2 instances, now struggles to provide extensive service coverage and detailed assessments for containerized applications, lacks continuous monitoring, and faces integration challenges.

By recognizing these limitations, this paper highlights the necessity for more robust, adaptive, and user-friendly security assessment tools that align with the complicated and dynamic nature of cloud environments. The proposed Security Vulnerability Assessment Tool is a step towards filling this crucial need in the domain of cloud security.

II. METHODS

A. Motivation

The rapid adoption of cloud platforms, notably AWS, Azure, and GCP, has equipped businesses with unprecedented scalability and operational flexibility. However, these benefits are accompanied by challenges. An abundance of security risks, predominantly stemming from misconfigurations, has led to significant data breaches and complex compliance issues.

Existing tools, such as AWS's Self-Service Security Assessment Solutions v2.0, offer partial solutions but fall short in areas such as real-time monitoring, comprehensive risk assessment, and user-specific customization. This gap underscores the urgent need for a more sophisticated tool capable of accurately identifying and mitigating vulnerabilities related to misconfigurations.

To address these vulnerabilities, businesses often resort to resources like AWS forums, whitepapers, and academic research. Although they provide valuable insights into common errors and misconfigurations, a systematic approach to identify, classify, and rectify these vulnerabilities is notably absent. The frequency of AWS-related data breaches underlines this problem, emphasizing the crucial need for our proposed tool.

A significant advancement in this domain is the adaptation of a classification system akin to Common Vulnerabilities and Exposures (CVE). This system aims to identify and categorize vulnerabilities, and deliver actionable insights for their mitigation, tailored to the specific operational contexts of users.

B. Research

The rise of cloud computing has made keeping data safe more complicated, especially with services like Amazon Web Services (AWS). Existing literature underscores the importance of exhaustive service coverage in security assessment tools, which is critical for ensuring that no aspect of the cloud environment remains unmonitored and susceptible to breaches. There have been instances when not having a full check-up led to issues, thereby reinforcing the imperative for comprehensive assessment solutions [8-25].

Parallel to the breadth of service coverage is the need for deep customization. Along with checking everything, these tools also need to customizable to fit each company's own way of doing things. If they are too fixed or rigid and cannot be adjusted, they might not catch all the potential security risks, neglecting the unique operational and architectural nuances of organizations, and therefore being less effective and potentially leading to configuration errors [26-30]. The stream of research in this area advocates for flexible, customizable tools that can adapt to an organization's specific cloud ecosystem.

Being able to continuously monitor potential problems is another critical requirement. Real-time monitoring emerges as another pivotal theme in the literature, positing that the dynamic nature of cloud environments necessitates continuous vigilance. Tools that only perform periodic scans, such as AWS's own tools, are not enough to manage constantly evolving threats.

Integration capabilities or the lack thereof also feature prominently in the literature. Studies illustrate that security tools which fail to integrate seamlessly with both AWS services and third-party solutions limit the scope of security measures and can leave blind spots within the security setup of organizations. The ability to define custom vulnerabilities and the timeliness of notifications are identified as areas where current tools fall short. Security tools must allow organizations to specify their own security rules and the need for prompt notifications to ensure swift response to potential threats.

The critical need for Security Vulnerability Assessment Tool is comprehensive, customizable, offers user-friendly, capable of advanced threat detection, integrates smoothly across cloud services, and provides timely alerts. This research aims to bridge these gaps by developing a tool tailored for AWS environments, setting a new benchmark for cloud management [31-35].

C. Approach, Methodology, and Design

The development of the Security Vulnerability Assessment Tool for AWS was based on a robust multi-faceted methodology. Initially, a comprehensive analysis of the current landscape of cloud security was undertaken, emphasizing the pervasive issue of misconfigurations in AWS. Recognizing the inadequacies of existing solutions, development pivoted towards the design of a solution specifically tailored for AWS environments.

The methodology begins with extensive literature review and consultation with both primary (e.g. AWS forums) and secondary (e.g. academic research, whitepapers) resources to gather an understanding of prevalent vulnerabilities. The critical step following this is the requirement analysis phase, which details the tool's requisite functionalities, laying the groundwork for subsequent prototyping. Leveraging AWS SDKs, particularly for Python (Boto3), an initial prototype of the tool was developed, focusing on AWS-specific misconfigurations.

Furthermore, as the tool's development progressed, seamless integration with other AWS services is prioritized, enhancing its usability and efficacy. Real-time monitoring and event-triggered assessments form the core of this approach. Lastly, in the spirit of comprehensive solution, the tool is developed to align with globally recognized vulnerability classification systems, like the CVE, to offer users actionable insights into potential security threats.

Throughout this methodology, the emphasis is on continuous learning, iterative development, and ensuring the tool remains adept at addressing the evolving challenges of AWS environments.

This application is an AWS Vulnerability Scanner and Dual List built using PyQt5. It is a GUI application designed to scan selected AWS services for vulnerabilities. The application provides a user-friendly interface, featuring a dual list selector for users to easily choose the AWS services they wish to scan. The scan results are displayed in real-time within the application window, allowing users to promptly address any identified vulnerabilities.

Features

- **Dark-Themed User Interface**: A sleek, user-friendly interface that reduces eye strain, ensuring user comfort during prolonged use.
- **Dual List Selector:** Enables users to effortlessly select AWS services for scanning.
- **Real-Time Scan Results**: Users can view the scanning process and results in real-time, facilitating immediate action for identified vulnerabilities.
- **AWS Service Support**: The application supports a wide range of AWS services, offering comprehensive vulnerability scanning.

I. User Interface (UI):

- Main Application Window: Serves as the central interaction point for users, providing access to the scanner's functions.
- Service List Widget: Displays the list of AWS services categorized for ease of use, like Compute, Storage, Database, Networking, Security, Identity & Compliance, Analytics, Application Integration, Deployment & Management and Others.
- Dual-list Widget: Enables a straightforward selection process for AWS services that need to be scanned, supporting both single and multiple actions.

II. Backend Services:

- Initialization Script: Utilizes PyQt5 for UI elements and initializes the application.
- AWS Service Integration: Leverages the Boto3 library for AWS API interactions, retrieving configuration details.
- Scan Engine: This is the core of the application, analyzing service configurations, comparing them to known vulnerability criteria, and pinpointing security risks.
- Scan Controller: Orchestrates the scanning process, including execution and real-time progress updates.

III. Result Storage:

This component is responsible for the temporary storage of scan outcomes, which are immediately displayed to the user. This feature facilitates the prompt assessment and remediation of identified security vulnerabilities. Additionally, the tool provides a feature to export the scan results to a text file. This allows users to save the results for later review or for documentation purposes, ensuring they have a record of the vulnerabilities that were found during the scan and can revisit the findings as needed.

IV. Communication & Integration:

- AWS CLI Configuration: Configures the necessary AWS credentials, enabling the tool to interact with the user's AWS ecosystem.
- Python Environment: Operates within Python 3.x, replying on PyQt5 and Boto3, among other libraries.

V. User Feedback & Control:

- Real-time Feedback Systems: Offers a dynamic view of scanning activity, showing vulnerabilities as they are detected in real time.
- Scan Control Buttons: Provides user control over the scanning process with "Scan" and "Stop Scan" functionalities.
- Result Viewer: Presents a digest of the scan, listing vulnerabilities along with remediation recommendations post-analysis.

VI. Security:

- Read-only Scan: Ensures a non-intrusive scan by reading service configurations without altering them.
- Safe Stop Mechanism: Allows interruption of the scan safely, with the tool summarizing findings up to the point of cessation.

VII. Target Environment:

The tool is engineered to support a broad array of AWS services, aiming to provide an exhaustive security assessment tool. It can scan core services addressing varied needs across AWS suites.

VIII. Services Covered:

Our Security Vulnerability Assessment Tool is designed to comprehensively scan a wide range of AWS services, categorizing them into distinct groups based on their functionality and use cases. This categorization not only aids in organized scanning but also ensures that a broad spectrum of AWS services is covered, mitigating the risk of overlooked vulnerabilities. Below is a detailed breakdown of the AWS service groups and the individual services within each group that our tool is capable of scanning:

Compute:

- Amazon EC2 (Elastic Compute Cloud): For assessing virtual server configurations.
- AWS Lambda: To check serverless computing setups.
- Amazon EC2 (Elastic Container Service): Focused on container orchestration.
- EC2 Auto Scaling: Evaluating scaling policies and configurations.
- AWS Elastic Beanstalk: Examining application deployment and management services.

Storage:

- Amazon S3 (Simple Storage Service): Scanning for bucket policies and access controls.
- Amazon Elastic Transcoder: Assessing media transcoding services.

Database:

- Amazon RDS (Relational Database Service): For database instance configurations.
- Amazon DynamoDB: NoSQL database, service checks.
- Amazon Redshift: Data warehouse service scanning.
- Amazon Athena: Serverless interactive query service evaluation.

Networking:

- Amazon VPC (Virtual Private Cloud): Verifying network configuration and security.
- Amazon Route 53: DNS service checks.
- Amazon CloudFront: Content delivery network service assessments.
- Elastic Load Balancing (ELB): Load balancer configuration checks.
- Amazon API Gateway: Evaluating API management.

Security Identity & Compliance:

- AWS Identity and Access Management (IAM): Scanning for user and role configurations.

- AWS Key Management Service (KMS): Key storage and management checks.
- AWS Secrets Manager: Security managing secrets.
- AWS WAF (Web Application Firewall): Firewall rule evaluations.
- Amazon Inspector: Security assessment service checks.
- Amazon Cognito: User identity and access configurations.

Analytics:

- Amazon CloudWatch: Monitoring service configuration checks.
- Amazon Elasticsearch Service: Elasticsearch cluster assessments.
- AWS Glue: Data integration service scanning.

Application Integrations:

- Amazon Simple Notification Service (SNS): Notification service configurations.
- Amazon Simple Queue Service (SQS): Message queuing service evaluations.
- AWS Step Functions: Workflow service checks.

Deployment & Management:

- AWS CloudFormation: Infrastructure as code service checks.
- AWS CodeDeploy: Automated deployment service assessments.
- AWS CloudTrail: Logging service evaluations.

Others:

- Amazon SWF (Simple Workflow Service): Workflow service for cloud applications.
- Amazon Elastic Container Registry (ECR): Docker container registry checks.

Each service within these categories is exhaustively scanned to identify potential misconfigurations that could lead to security vulnerabilities. Our tool's scanning methods are specifically tailored to the nuances and typical configuration pitfalls associated with each AWS service, ensuring both depth and breadth in security assessments. Figure 1(a) shows the user interface of this tool, Fig. 1(b) shows the user interface after selecting scans (EC2, S3 and VPC), Fig. 1(c) shows an overview of AWS services covered by the Security Vulnerability Tool, and Fig. 1(d) shows the architectural design of this tool.

Fig. 1(a) User interface of the AWS Security Vulnerability Tool.

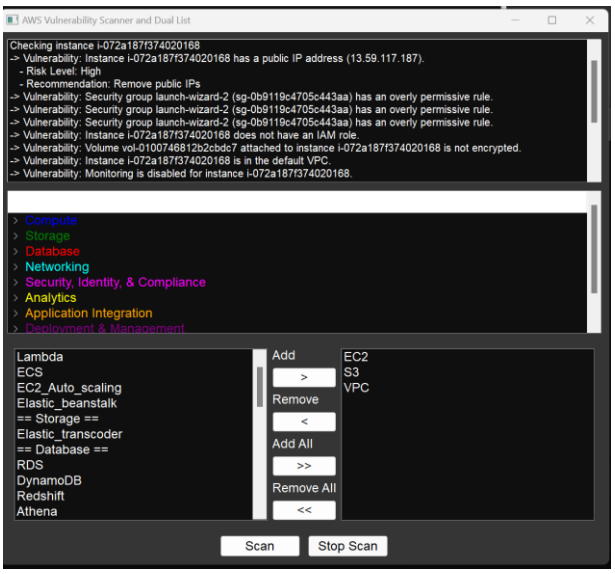


Fig. 1(b) User interface of the AWS Security Vulnerability Tool after selecting scans (EC2, S3 and VPC).

Security Vulnerability Assessment Tools: AWS Services

[Compute Services]
- Amazon EC2 → Virtual server configurations
- AWS Lambda → Serverless computing setups
- Amazon ECS → Container orchestration
- EC2 Auto Scaling → Scaling policies
- AWS Elastic Beanstalk → Application deployment

[Storage Services]
- Amazon S3 → Bucket policies, access controls
- Amazon Elastic Transcoder → Media transcoding

[Database Services]
- Amazon RDS → Database instances
- Amazon DynamoDB → NoSQL service checks
- Amazon Redshift → Data warehouse scanning
- Amazon Athena → Serverless queries

[Networking Services]
- Amazon VPC → Network configuration
- Amazon Route 53 → DNS checks
- Amazon CloudFront → CDN assessments
- Elastic Load Balancing → Load balancing
- Amazon API Gateway → API management

[Security, Identity & Compliance]
- AWS IAM → User/role configurations
- AWS KMS → Key management
- AWS Secrets Manager → Managing secrets
- AWS WAF → Firewall rules
- Amazon Inspector → Security assessments
- Amazon Cognito → User identity/access

[Analytics Services]
- Amazon CloudWatch → Monitoring services
- Amazon Elasticsearch → Cluster assessments
- AWS Glue → Data integration

[Application Integration Services]
- Amazon SNS → Notification services
- Amazon SQS → Message queuing
- AWS Step Functions → Workflow checks

[Deployment & Management Services]
- AWS CloudFormation → Infrastructure code
- AWS CodeDeploy → Automated deployment
- AWS CloudTrail → Logging services

[Other Services]
- Amazon SWF → Workflow services
- Amazon ECR → Docker registry checks

Fig. 1(c): Overview of AWS services covered by the Security Vulnerability Tool.

Types of Misconfigurations Detected:

For each detected vulnerability, the tool provides a risk assessment based on factors like potential exposure, data security implications and compliance requirements. The risk levels (e.g., High, Medium, Low) and recommendations are determined based on industry standards and best practices, potentially integrating CVSS (Common Vulnerability Scoring System) scores for a more standardized risk assessment. This approach helps prioritize remediation efforts based on the severity of the vulnerability [36].

CVSS scoring:

The tool creates a custom scoring system for AWS vulnerabilities [34-36]:

1. Scoring Process:
   - Base Score: Calculated considering impact and exploitability metrics. It is the intrinsic and fundamental score associated with a vulnerability.
   - Temporal Score: Refines the Base Score over time, considering the current exploitability remediation level, and confidence in the reported vulnerability details.
   - Environmental Score: Customizes the Temporal Score by accounting for the potential impact on a specific organization, factoring in security controls, and how critical affected systems are.
2. Scoring Metrics:
   2.1. Exploitability Metrics:
   - Access Vector (AV)
   - Access Complexity (AC)
   - Authentication (Au)

   2.2. Impact Metrics:
   - Confidentiality Impact (C)
   - Integrity Impact (I)
   - Availability Impact (A)

   2.3 Temporal Metrics:
   - Exploitability (E)
   - Remediation Level (RL)
   - Report Confidence (RC)

   2.4 Environmental Metrics:
   - Collateral Damage Potential (CDP)
   - Target Distribution (TD)
   - Confidentiality, Integrity, and Availability Requirement (CR, IR, AR)

IX. Deployment Strategy:

A. The AWS Vulnerability Scanner is designed as a standalone GUI application that users can deploy within their local environment. It operates by interfacing with the AWS environment through secure API calls facilitated by AWS CLI and Boto3. The approach ensures that users can conduct vulnerability assessments from their workstations with real-time insights and immediate visibility into potential security risks. The application's deployment is straightforward, requiring a Python environment and the installation of necessary libraries.

B. Docker deployment: To simplify deployment and ensure a consistent environment regardless of the user's local setup, the tool also offers a Docker-based deployment option. Using Docker, the application is packaged along with its dependencies, eliminating discrepancies that might arise from different local environments. The deployment strategy for AWS Vulnerability Scanner is designed to be flexible, catering to both users who prefer a local Python environment and those who opt for a Docker-based solution. The Docker deployment, in particular, offers a streamlined, consistent, and isolated environment, enhancing the tool's usability and reliability across different systems.

X. User Profiles and Needs:

Our primary user base consists of cloud administrators and security professionals, DevOps teams, and small and medium-sized businesses (SMBs) and startups.

Cloud Administrators and Security Professionals:

- Require a tool that provides comprehensive insights into the current security situation.
- Need capabilities for detailed and customizable security assessments.
- Value real-time alerts and integration with existing security workflows.

DevOps Teams:

- Seek automated and continuous security assessment solutions.
- Benefit from integration with CI/CD pipelines for continuous compliance.
- Prefer tools that offer actionable insights and recommendations.

SMBs and Startups:

- Require cost-effective solutions for cloud security.
- Need simple, user-friendly interfaces for ease of use without deep technical expertise.
- Value comprehensive coverage without the need for extensive configurations.

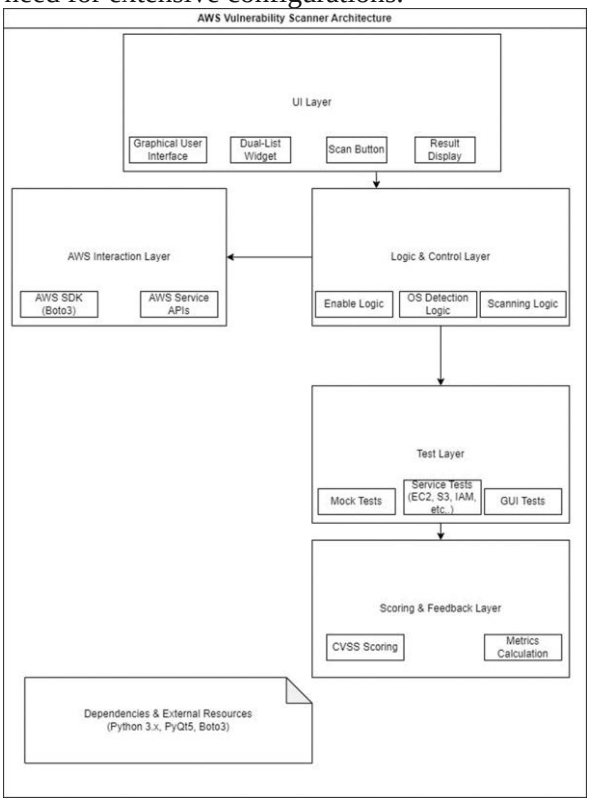


Fig. 1(d) Architectural design.

III. TESTING

Testing Environment

The testing environment for this tool includes both simulated AWS environments, leveraging the 'moto' library to mock AWS services, and a graphical user interface (GUI) application developed using PyQt5. The 'moto' library allows simulating AWS resources without incurring costs or side effects in a real cloud environment. This approach ensures that the tests can run in isolation making them reproducible and independent of actual AWS configurations.

Testing Methodology:

The testing methodology employs a suite of unit and integration tests, written using the 'pytest' framework, to validate both the functionality of the AWS interactions and the GUI elements. The tests are designed to simulate user interactions with the GUI, as well as the application's responses to various AWS service scenarios, such as creating resources, managing permissions, and detecting configurations [37-38].

For GUI elements, user actions like button clicks and list item selections are simulated, as well as the expected state changes within the application. The tests include verifying the enablement state of buttons before and after actions, the population of list widgets with AWS service names, and the response of the application to user interactions, such as moving items between lists.

The AWS service mocking involves simulating different AWS resources and configurations. For example, mock EC2 instances were created with public IPs and open security groups to test the scanning procedures for potential vulnerabilities. CloudWatch alarms, S3 buckets with various access policies, and lambda functions with specific IAM roles were also simulated.

All 23 tests passed, indicating that the application's components interact correctly with the mocked AWS services and the GUI elements behave as expected. The test results are indicative of a robust testing strategy, providing the tool's effectiveness in the simulated environment.

The Mock Testing script consists of unit tests for a cloud security scanning application, particularly focusing on AWS services as shown in Fig. 2.

1. Libraries and Resources Initialization
   - Libraries: Importation of required libraries including 'pytest', 'boto3', 'moto', 'zipfile', and 'PyQt5'.
   - Resource Creation: Creation of a ZIP file containing a lambda function, essential for testing AWS Lambda.
2. Mocking and Testing Utility:
   - MockEmitter Class: A utility class to mock the emission of messages during the testing phase, used to capture and store messages emitted during tests.
   - Application Instance: Instantiation of 'Qapplication' for testing GUI elements.
3. Testing AWS Services and GUI Elements:
   The testing process is extensive, covering a range of AWS services and the application's GUI elements, and is structured into several function tests.
   - GUI Initialization Tests: To confirm that the main window and other GUI elements like dual list widgets are initialized correctly.
   - Service Selection Test: Uses 'moto' to mock AWS EC2 and IAM, and testing if the scan can identify specific vulnerabilities.
   - EC2 Scan Test: Uses 'moto' to mock AWS EC2 and IAM, and tests if the scan can identify specific vulnerabilities.
   - S3 Scan Test: Mocks S3 and tests if the scanner identifies vulnerabilities related to bucket permissions.
   - IAM Scan Test: Evaluates if the scan identifies vulnerabilities in AWS IAM configurations.
   - SQS Scan Test: Checks for vulnerabilities in SQS, particularly focusing on public access.
   - SNS Scan Test: Evaluates the tool's capability to identify public access vulnerabilities in SNS messages.
   - Lambda Scan Test: Confirms that the scanning tool can evaluate AWS Lambda functions, checking for permissions and configurations.
   - ElasticSearch Scan Test: Tests the tool's ability to scan and evaluate AWS ElasticSearch configurations for vulnerabilities.
   - CloudFront Scan Test: Evaluates the scanner's ability to identify CloudFront distributions without a default root object.
4. Scanning Procedures and Button States:
   - Scanning Procedure: Tests to ensure the scanning procedure operates and populates the result box correctly.
   - Button State Tests: Confirms that button states (enabled/disabled) change appropriately during and after scans.
5. AWS VPC and ELB Tests:
   - VPC Creation Test: Ensures that a VPS is configured with correct CIDR block.
   - ELB Scan Test: A test function to check vulnerabilities in Elastic Load Balancer configurations.
6. CloudWatch Alarms Test:
   - Alarm Configuration Test: Confirms the CloudWatch alarm configurations and ensures it does not falsely identify the absence of alarms.
7. Interactive Tests:
   - Scan Interruption Test: Confirms that the scan can interrupt/stop and evaluate the state of the scan and stop buttons before, during, and after a scan.
   - Result Box Content Test: Confirms the result box is populated after the scan.
8. Emission Tests:
   - Signal Emission Test: Ensures the scan thread emits signals as expected, validating the interactive feedback mechanism of the tool.

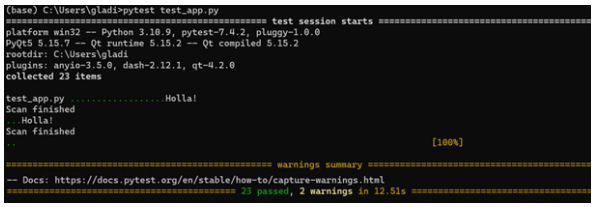


Fig. 2 Testing result

IV. RESULTS

Findings from Testing: The results demonstrate the tool's effectiveness in detecting misconfigurations, with a high percentage of true positives and negatives.

Comparison with Existing Tools:

AWS Inspector vs. This Tool

Integration and Ecosystem:

- AWS Inspector is closely integrated with the AWS ecosystem, offering specialized insights for AWS services.
- This tool interfaces with a diverse range of AWS services, potentially providing broader functionality than AWS Inspector.

User Interface:

- AWS Inspector utilizes the AWS Management Console, offering a familiar interface to those accustomed to AWS.
- This tool boasts a dark-themed UI and a dual list selector, potentially offering a more intuitive and visually appealing experience.

Customizability:

- AWS Inspector follows standard security assessments based on common practices and compliance standards.
- This tool allows more tailored scans and specific vulnerability checks, offering enhanced adaptability to individual user needs.

Real-Time Feedback:

- AWS Inspector conducts scheduled assessments, with reports delivered after analysis.
- This tool provides immediate scan results, facilitating swift identification and action on vulnerabilities.

AWS Self-Service Security Assessment Solution v2.0 vs. This Tool

Scope and Depth of Assessment:

- The AWS Self-Service Security Assessment Solutions v2.0 includes a comprehensive set of tools, but many require manual effort and in-depth AWS knowledge.
- This tool automates vulnerability scanning, offering a more streamlined approach for users.

Ease of Use:

- AWS Solutions might be more complex for users less familiar with AWS services.
- This tool's user-friendly UI and instant feedback make it accessible to a wider audience, including those without a strong technical background.

Scalability:

- AWS Solutions scales with AWS environments, but scalability can vary between tools within the Solutions.
- The scalability of this tool in handling large AWS environments is crucial, especially in comparison to AWS-native tools.

Cost:

- AWS Solutions follows the AWS pricing model, which might incur usage-based charges.
- If this tool is free or has a different pricing model, it could be a more economical option for users.

Customizability and Extensibility:

- AWS Solutions offers customization within a standardized framework.
- This tool could provide greater customizability, such as adding new scanning capabilities or integrating with external tools.
- **Scalability to Other Cloud Platforms**: Initially designed for AWS, this tool is engineered for seamless scalability to other cloud platforms, such as Azure. It allows for easy integration and adaptation of scanning capabilities to Azure services, enabling a unified approach to cloud security across multiple cloud environments. This scalability ensures that organizations using multi-cloud strategies can maintain a consistent security posture across all their cloud assets. (Fig. 3).

Fig. 3. Scalability to the Azure Platform: Extending Capabilities

TABLE 1. COMPARSION TABLE

Comparison Criteria	AWS Inspector and AWS Self-Service Security	This Tool
Integration and Ecosystem	• Closely integrated with AWS • Specialized insights for AWS services	• Interfaces with a diverse range of AWS services • Potentially broader functionality
User Interface	• Utilizes the AWS Management Console	• Dark-themed UI • Dual list selector

	• Familiar interface for AWS users	• Potentially more intuitive and visually appealing • Tailored scans and specific vulnerability checks
Customizability	• Standard security assessments • Based on common practices and standards	• Enhanced adaptability to user needs
Real-time Feedback	• Scheduled assessments with post-analysis reports	• Immediate scan results • Swift identification and action on vulnerabilities • Automates vulnerability scanning
Scope and Depth of Assessment	• Comprehensive toolset requiring manual	• More streamlined approach
Ease of Use	• May be complex for less familiar users	• User-friendly UI and instant feedback • Accessible to a wider audience, including non-technical users
Scalability	• Scales with AWS environments • Scalability varies across tools	• Handles large AWS environments • Crucial scalability in comparison to AWS-native tools
Cost	• Usage-based charges per AWS pricing model	• Potentially more economical if free or differently priced
Customizability and Extensibility	• Customization within a standardized framework	• Great customizability and extensibility • New scanning capabilities, external tool integration

To conclude, while AWS Inspector and AWS Self-Service Security Assessment Solutions v2.0 provide integrated and comprehensive security solutions for AWS, this tool differentiates itself with its user-centric interface, immediate feedback capabilities and potential for greater customization and flexibility in scanning various AWS services.

Case Studies:

Case Study 1: Extracting Bzure Price API Data: A newly onboarded engineer is assigned the task of daily extraction of Bzure price API data. Using an EC2 instance, they are to perform API calls, transform the acquired data, and archive it in an S3 bucket. Upon completion of these tasks, stakeholders are to be informed via SNS notification. Figures 4(a) and 4(b) present a pipeline and systematic summary of Case Study 1, respectively.

Tests Conducted:

EC2 Instances: Tested EC2 instance for potential security threats by simulating 100 instances with public IP assignments, VPCs, subnets and security groups.

S3 Bucket Misconfigurations: Tested S3 scripts among 120 instances including write access, public access, versioning, encryption, logging, read access and bucket policies. Proper configuration of S3 buckets is crucial for data integrity and security. The test aims to ensure that the bucket where the Bzure price data is archived is private, encrypted and has logging enabled to track any access or changes, which aligns with data security best practices.

SNS Misconfigurations: Tested SMS topics by checking 100 instances for common misconfigurations. This ensures that only authorized stakeholders receive notifications and that the notification system itself does not introduce vulnerabilities.

Misconfiguration Tests and Results:

EC2 Instances (100 tested):

- True Positives (TP): 50% (instances correctly identified with misconfigurations).
- True Negatives (TN): 50% (instances correctly identified with misconfigurations).

S3 Buckets (120 tested):

- True Positives (TP): 80% (buckets correctly identified with misconfigurations).
- True Negatives (TN): 40% (buckets correctly identified without misconfigurations).

SNS Topics (100 tested):

- True Positives (TP): 50% (topics correctly identified with misconfigurations).
- True Negatives (TN): 50 % (topics correctly identified without misconfigurations).

Total across all services (320 instances):

- True Positives: 56.98% (instances correctly identified with misconfigurations).
- True Negatives: 43.02% (instances correctly identified without misconfigurations)

Case Study 1 emphasizes the importance of proper AWS configuration and the need for ongoing security checks. It also reflects the effectiveness of the testing tool and the necessity for continuous education and improvement in cloud management practices, particularly for newly onboarded engineers handling critical data.

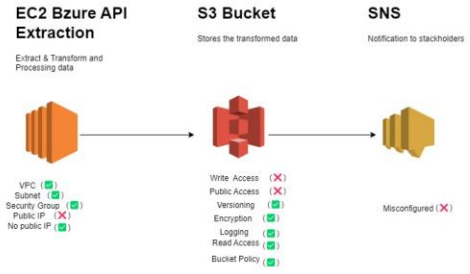


Fig. 4(a). Pipeline of Case Study 1.

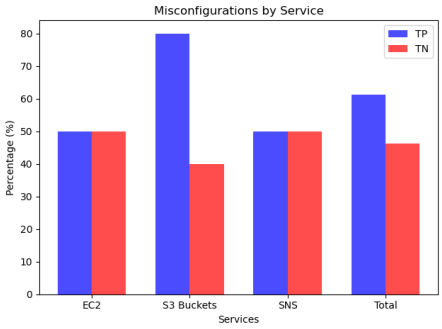


Fig. 4(b). Systematic summary of Case Study 1.

Case Study 2: AWS Misconfiguration in Rainfall Data Extraction

A junior engineer is tasked with collecting hourly rainfall data and, due to inexperience, incurs multiple misconfigurations across AWS services. Figures 5(a) and 5(b) present a pipeline and systematic summary of Case Study 2, respectively.

Tests Conducted:

- Lambda Function Creation: A lambda function was created with an overprivileged role, which is a security concern because it could lead to unauthorized access or actions.
- Glue Jobs' role: Glue jobs were created with an overprivileged role, indicating a misconfiguration in this area.
- S3 Data Storage and Validation: The test involved storing and validating multiple S3 objects to ensure data integrity.
- Redshift Cluster Encryption and Logging: The test checked whether the Redshift data warehouse cluster was encrypted, which is critical to protect data at rest. The test verified if logging was enabled for the Redshift cluster, which is important for monitoring and security auditing.

Misconfiguration Tests and Results:

Out of 200 instances tested across the services:

- True Positives: 24.72% were correctly identified as misconfigured.
- True Negatives: 75.28% were correctly identified as well-configured.
- False Positives & Negatives: There were no inaccuracies reported in the test results.

This case study underscores the necessity of security awareness in cloud operations, especially for engineers who are new to AWS. It shows a proactive approach in using automated tests to identify potential security risks, which is best practice in cloud security management. It also suggests that while the junior engineer had made misconfigurations, there was an effective safety net in place to catch these before they became critical issues. A startup can use these findings to refine their onboarding and training, ensuring engineers are better prepared to manage the security of AWS resources.

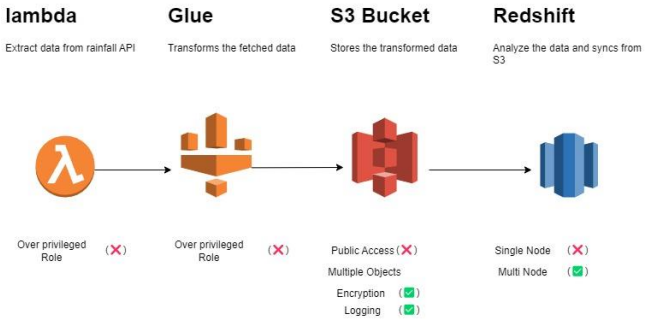


Fig. 5(a). Pipeline of Case Study 2.

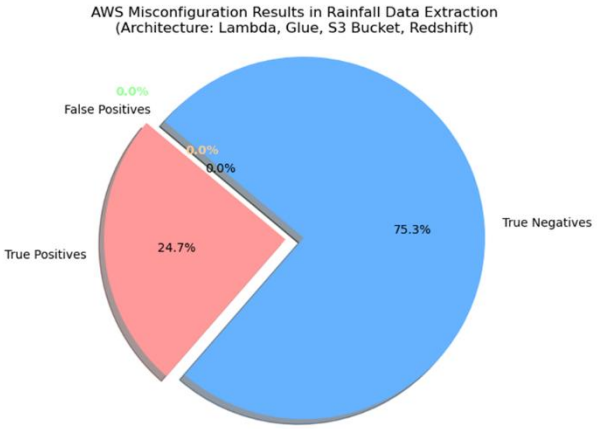


Fig. 5(b). Systematic summary of Case Study 2.

V. Limitations and Future Work

The Security Vulnerability Assessment Tool, designed primarily for AWS, shows promise but also has some limitations that can be improved. Future updates can include enhancing the tool's ability to conduct behavioral analysis. By better understanding the complex behaviors within cloud infrastructure, the tool could detect subtle security issues not immediately obvious from setups alone. As security threats are ever-changing, there is a need to create new ways to quickly identify and stop new kinds of cyber-attacks. The tool's reliance on AWS's own systems means it could face challenges if those systems change, suggesting a need contingency plans to keep the tool functioning without issues. Additional improvements to the tool could include leveraging artificial intelligence to predict security risks before they happen using advanced algorithms. Making the tool open source could also make it more powerful, tapping into a wider pool of knowledge by allowing developers to improve and extend the tool's functions. Lastly, with many organizations using more than one cloud provider, adapting this tool to work with others like Azure could help maintain robust security across different platforms.

VI. Conclusions

This extensive research and testing have led to critical insights into the configuration and security of AWS resources. The two case studies, focusing on the extraction of Bzure price API data and rainfall data collection, have

demonstrated the effectiveness of this tool in identifying misconfigurations. This has been quantitatively measured through a high percentage of true positives and true negatives. The tool has shown broader functionality than AWS Inspector by interfacing with a wide range of AWS services and providing immediate feedback, which is crucial for timely vulnerability management.

The findings reinforce the paramount importance of stringent cloud security practices. Misconfigurations in cloud services, as seen with the EC2 instances and S3 buckets, pose significant risks. The tool's capability to identify and alert to such vulnerabilities promptly helps mitigate potential threats. The immediate feedback mechanism empowers users to act swiftly, enhancing the security posture within AWS environments.

The tool is particularly relevant for cloud engineers and security teams who are responsible for maintaining the security and integrity of cloud resources. It addresses specific challenges such as event-driven detection of misconfigurations, a user-friendly interface for easier navigation, and customizability for tailored security assessments. Users without a strong technical background can also benefit from the intuitive UI and simplified feedback mechanism.

Moving forward, future objectives include expanding the tool's capabilities to encompass more AWS services and integrate with additional security frameworks. Machine learning techniques will also be introduced to predict potential misconfigurations based on usage patterns. Furthermore, the potential for automated remediation actions based on the identified vulnerabilities will be explored.

ACKNOWLEDGMENT

This work would not have been possible without the support and guidance of several distinguished individuals and teams.

I extend my deepest gratitude to Professor Mustaque Ahamad, whose expertise and mentorship were instrumental in shaping the research and its outcomes. His insightful comments and unwavering support throughout the project were invaluable.

My sincere thanks also go to TA Subhiksha Ramanathan for her astute observations and detailed feedback, which significantly enhanced the quality of this work.

Furthermore, I am grateful for the collaboration and insights provided by the AWS Marketing Intelligence Team. Their expertise in marketing intelligence has been a vital contribution to this research.

Their collective wisdom and support have been fundamental to the success of this endeavor.

REFERENCES

[1] S. Hollister, "Massive Capital One breach exposes personal info of 100 million Americans," *The Verge*, Jul. 29, 2019. https://www.theverge.com/2019/7/29/20746493/massive-capital-one-breach-exposes-personal-info-of-100-million-americans.

[2] "Turkish Based Airline's Sensitive EFB Data Leaked," *SafetyDetectives*. https://www.safetydetectives.com/news/pegasus-leak-report/#:~:text=Almost%2023%20million%20files%20were (accessed Sep. 26, 2023).

[3] R. M. published, "Data breach exposes millions of seniors' data," *ITPro*, Aug. 09, 2021. https://www.itpro.com/data-insights/big-data/360525/data-breach-exposes-details-on-millions-of-us-seniors (accessed Sep. 26, 2023).

[4] P. Paganini, "Over 80 US Municipalities ' Sensitive Information, Including Resident's Personal Data, Left Vulnerable in Massive Data Breach," *Security Affairs*, Jul. 23, 2021. https://securityaffairs.com/120477/data-breach/us-municipalities-data-breach.html (accessed Nov. 08, 2023).

[5] "Data Breach: Hundreds of Thousands of Customers' Personal Information Exposed," *WizCase*. https://www.wizcase.com/blog/cosmolog-breach-report/ (accessed Nov. 08, 2023).

[6] M. X. Heiligenstein, "Amazon Web Services (AWS) Data Breach Timeline," *Firewall Times*, Aug. 27, 2021. https://firewalltimes.com/amazon-web-services-data-breach-timeline/.

[7] "Report: Hotel Reservation Platform Leaves Millions of People Exposed in Massive Data Breach," *Website Planet*, Nov. 06, 2020. https://www.websiteplanet.com/blog/prestige-soft-breach-report/

[8] J. Guffey and Y. Li, "Cloud Service Misconfigurations: Emerging Threats, Enterprise Data Breaches and Solutions," *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*, Mar. 2023, doi: https://doi.org/10.1109/ccwc57344.2023.10099296.

[9] D. K. Saini, K. Kumar, and P. Gupta, "Security Issues in IoT and Cloud Computing Service Models with Suggested Solutions," *Security and Communication Networks*, vol. 2022, pp. 1–9, Apr. 2022, doi: https://doi.org/10.1155/2022/4943225.

[10] A. Verdet, M. Hamdaqa, L. Da Silva, and F. Khomh, "Exploring Security Practices in Infrastructure as Code: An Empirical Study," *arXiv.org*, Aug. 07, 2023. https://arxiv.org/abs/2308.03952 (accessed Aug. 25, 2023).

[11] A. S. Muhammed and D. Ucuz, "Comparison of the IOT platform vendors, Microsoft Azure, Amazon Web Services, and google cloud, from users' perspectives," *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*, 2020. doi:10.1109/isdfs49300.2020.9116254.

[12] Satyavathi Divadari, J. Surya Prasad, and P. B. Honnavalli, "Managing Data Protection and Privacy on Cloud," pp. 383–396, Jan. 2023, doi: https://doi.org/10.1007/978-981-19-6088-8_33.

[13] S. Devi and T. S. Bharti, "Study of Architecture and Issues in Services of Cloud Computing," *IEEE Xplore*, Dec. 01, 2021. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9725679 (accessed Aug. 12, 2023).

[14] G. Sagar and V. Syrovatskyi, "Cloud: On Demand Computing Resources for Scale and Speed," *Technical Building Blocks*, pp. 53–104, 2022, doi: https://doi.org/10.1007/978-1-4842-8658-6_2.

[15] M. Jartelius, "The 2020 Data Breach Investigations Report – a CSO's perspective," *Network Security*, vol. 2020, no. 7, pp. 9–12, Jul. 2020, doi: https://doi.org/10.1016/s1353-4858(20)30079-9.

[16] D. Chen, M. M. Chowdhury, and S. Latif, "Data Breaches in Corporate Setting," *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, Oct. 2021, doi: https://doi.org/10.1109/iceccme52200.2021.9590974.

[17] S. Mishra, M. Kumar, N. Singh, and S. Dwivedi, "A Survey on AWS Cloud Computing Security Challenges & Solutions," *IEEE Xplore*, May 01, 2022. https://ieeexplore.ieee.org/abstract/document/9788254

[18] S. An *et al.*, "CloudSafe: A Tool for an Automated Security Analysis for Cloud Computing," *IEEE Xplore*, Aug. 01, 2019. https://ieeexplore.ieee.org/abstract/document/8887392 (accessed Sep. 01, 2023).

[19] S. An, A. Leung, J. B. Hong, T. Eom, and J. S. Park, "Toward Automated Security Analysis and Enforcement for Cloud Computing Using Graphical Models for Security," *IEEE Access*, vol. 10, pp. 75117–75134, 2022, doi: https://doi.org/10.1109/access.2022.3190545.

[20] ISO, "ISO/IEC 27001 standard – information security management systems," *ISO*, 2022. https://www.iso.org/standard/27001

[21] R. Python, "Python and PyQt: Building a GUI Desktop Calculator – Real Python," *realpython.com*. https://realpython.com/python-pyqt-gui-calculator/.

[22] “Graphical User Interfaces with Tk — Python 3.7.4 documentation,” *Python.org*, 2019. https://docs.python.org/3/library/tk.html.

[23] Docker, “Docker Documentation,” *Docker Documentation*, Oct. 31, 2019. https://docs.docker.com/.

[24] “How to Proactively Detect and Repair Common Misconfigurations on AWS Using AvailabilityGuard NXG,” *Amazon Web Services*, Dec. 17, 2019. https://aws.amazon.com/blogs/apn/how-to-proactively-detect-and-repair-common-misconfigurations-on-aws-using-availabilityguard-nxg/

[25] “AWS Well-Architected - Build secure, efficient cloud applications,” *Amazon Web Services, Inc.* https://aws.amazon.com/architecture/well-architected/?wa-lens-whitepapers.sort-by=item.additionalFields.sortDate&wa-lens-whitepapers.sort-order=desc&wa-guidance-whitepapers.sort-by=item.additionalFields.sortDate&wa-guidance-whitepapers.sort-order=desc.

[26] “AWS Whitepapers & Guides,” *Amazon Web Services, Inc.* https://aws.amazon.com/whitepapers/?whitepapers-main.sort-by=item.additionalFields.sortDate&whitepapers-main.sort-order=desc&awsf.whitepapers-content-type=.

[27] Amazon, “AWS Documentation,” *Amazon.com*, 2019. https://docs.aws.amazon.com/.

[28] “AWS Security Blog,” *Amazon.com*, Nov. 06, 2019. https://aws.amazon.com/blogs/security/.

[29] “What’s the Real Cost of an AWS Misconfiguration?,” *shardsecure.com*, Mar. 01, 2023. https://shardsecure.com/blog/real-cost-aws-misconfiguration (accessed Sep. 26, 2023).

[30] “Common Vulnerability Scoring System Version 3.1 Calculator,” *FIRST — Forum of Incident Response and Security Teams*. https://www.first.org/cvss/calculator/3.1#CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:L (accessed Sep. 29, 2023).

[31] *Amazon.com*, 2023. https://docs.aws.amazon.com/inspector/ (accessed Oct. 12, 2023).

[32] “OWASP RISK RATING CALCULATOR,” *owasp-risk-rating.com*. https://owasp-risk-rating.com/

[33] “Moto: Mock AWS Services — Moto 4.2.6.dev documentation,” *docs.getmoto.org*. http://docs.getmoto.org/en/latest/ (accessed Oct. 12, 2023).

[34] Imperva, “What is CVE and CVSS | Vulnerability Scoring Explained | Imperva,” *Learning Center*. https://www.imperva.com/learn/application-security/cve-cvss-vulnerability/.

[35] NIST, “NVD - Vulnerability Metrics,” *Nist.gov*, 2019. https://nvd.nist.gov/vuln-metrics/cvss.

[36] “NVD - CVSS v2 Calculator,” *nvd.nist.gov*. https://nvd.nist.gov/vuln-metrics/cvss/v2-calculator.

[37] “pytest: helps you write better programs — pytest documentation,” *docs.pytest.org*. https://docs.pytest.org/en/7.4.x/

[38] “Pytest Tutorial,” *www.tutorialspoint.com*. https://www.tutorialspoint.com/pytest/index.htm (accessed Nov. 29, 2023).

Appendix A

AWS Vulnerability Scanner User Guide

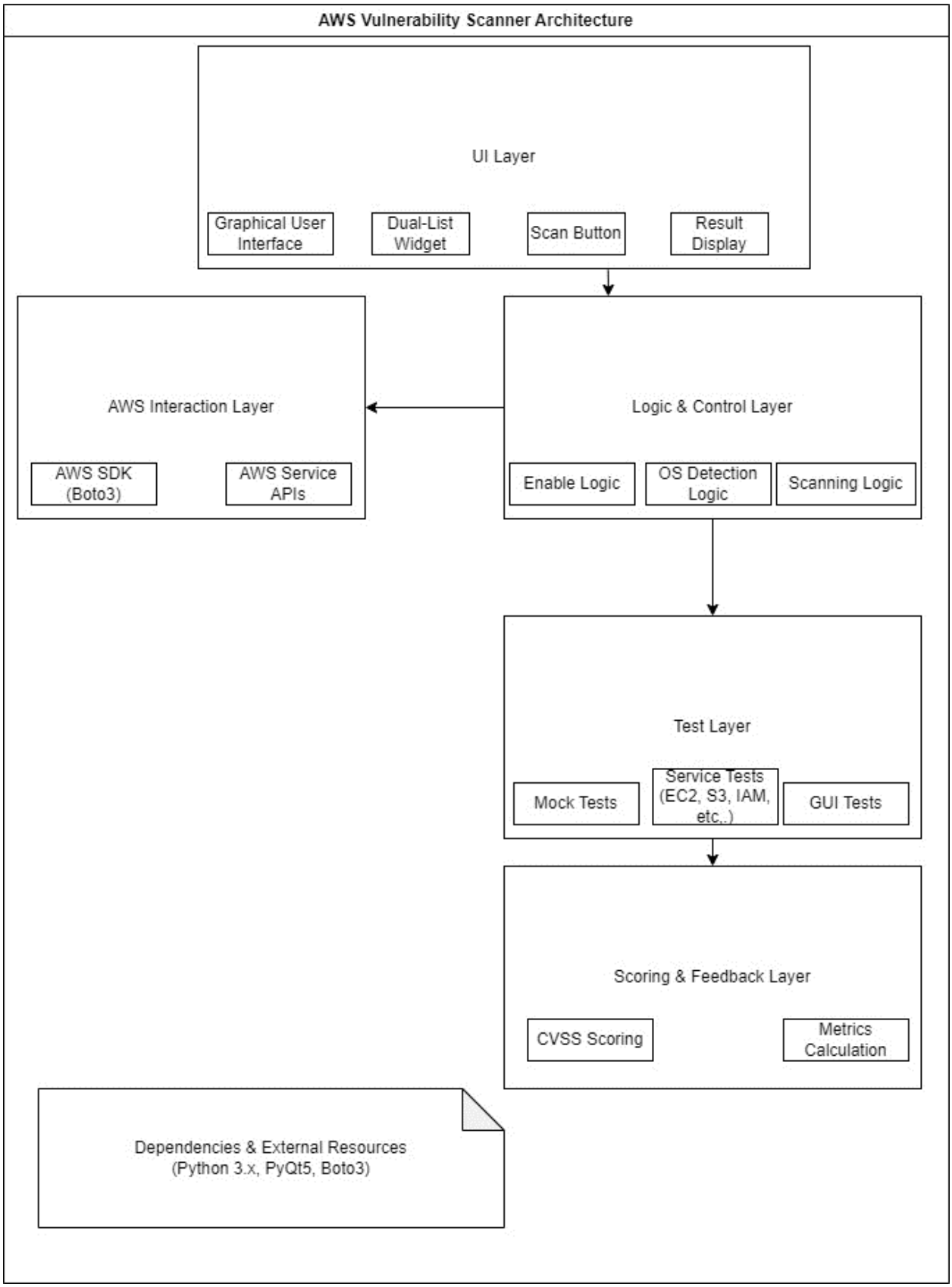
Introduction

The AWS Vulnerability Scanner is a specialized tool designed to help AWS users identify and diagnose potential misconfigurations and vulnerabilities in their AWS environment. With the proliferation of AWS services, ensuring proper security configurations is paramount. This tool provides a user-friendly interface, enabling users to scan a selection of AWS services and receive immediate feedback on potential areas of concern.

Architecture of the AWS Vulnerability Scanner:

- **User Interface (UI):**
    - Main Application Window: Acts as the primary entry point for users to interact with the scanner.
    - Service List Widget: Displays available AWS services grouped by category (e.g., Compute, Storage, Database).
    - Dual-list Widget: Allows users to select AWS services they wish to scan. Supports individual and bulk selection/deselection.
- **Backend Services:**
    - Initialization Script: Helps launch the AWS Vulnerability Scanner application. Uses the PyQt5 library for UI rendering.
    - AWS Service Integration: Uses the Boto3 library to interface with AWS services and fetch configuration details.
    - Scan Engine: The core component that processes AWS service configurations, checks against vulnerability definitions, and identifies potential threats.
    - Scan Controller: Manages the scan lifecycle, including starting, stopping, and monitoring the scan progress.
- **Database & Storage:**
    - Result Store: Temporarily stores the scan results for display in the result box. Users can review potential vulnerabilities and misconfigurations.
- **Communication & Integration:**
    - AWS CLI Configuration: Ensures the application has the necessary credentials to interact with the user's AWS environment.
    - Python Environment: Runs on Python 3.x and requires specific libraries like PyQt5 and Boto3.
- **User Feedback & Control:**
    - Real-time Feedback System: Provides users with a live view of the scanning process, including identified vulnerabilities.
    - Scan Control Buttons: Includes "Scan" and "Stop Scan" buttons for user control over the scanning process.
    - Result Viewer: Displays a post-scan summary of vulnerabilities, suggestions, or findings for users to take corrective actions.
- **Security:**
    - Read-only Scan: The tool only reads configurations from AWS services without making any changes, ensuring safety.
    - Safe Stop Mechanism: Users can halt the scanning process anytime, and the system provides a summary up to the stopped point.

User Interface



Architecture Design Diagram

System Requirements

1. Python 3.x
2. PyQt5 library
3. Boto3 library

Setup

1. Ensure that the AWS CLI is configured with the appropriate credentials and default region.
2. Install the required Python libraries if not already installed.
3. Run the provided Python script to launch the AWS Vulnerability Scanner application.

How to Use the AWS Vulnerability Scanner:

1. Launch the Application
   After running the script, the main window of the application will appear.
2. Explore Available AWS Services
   a. On the left side, you will see a list of AWS services that you can scan.
   b. Each service is grouped by its category for better clarity, e.g. Compute, Storage, Database.
3. Select AWS Services for Scanning
   a. Using the dual-list widget, you can:
      i. Select individual services to move them to the "Selected list by clicking on the ">" button.
      ii. Move services back to the "Available" list using the "<" button.
      iii. Move all services to the "Selected" list using the ">>" button.
      iv. Return all services to the "Available" list using the "<<" button.
   b. Ensure you have selected the AWS services you wish to scan.
4. Start the Scan
   Click the "Scan" button to initiate the vulnerability scanning process. Once the scan begins:
   a. The Scan button will be disabled.
   b. The Stop Scan button will become active.
5. Monitor the Scan
   a. The application will provide real-time feedback in the result box at the bottom.
   b. Here you will see potential vulnerabilities or misconfigurations identified for each AWS service.
6. Stop the Scan (if needed)
   If for any reason you wish to stop the scanning process, click the "Stop Scan" button. This will halt the scan, and the application will provide a summary of the results up to that point.
7. Review Results
   Once the scan completes:
   a. Review the results in the result box to identify potential areas of concern.
   b. Take note of any suggestions or findings to rectify them in your AWS environment.

Conclusion

The AWS Vulnerability Scanner is an invaluable tool for AWS users aiming to secure their environments. Regular use will ensure that your configurations align with best practices and that potential vulnerabilities are identified and rectified in a timely manner. Happy scanning!

Dockerizing the AWS Vulnerability Scanner:

Introduction

This guide outlines the process to set up and run a PyQt application inside a Docker container and access it using a VNC Viewer. This approach is beneficial for running GUI applications in isolated environments.

Prerequisites

- Docker installed on your system.
- VNC Viewer installed for accessing the GUI.
- The Dockerfile and the application source code.

Step 1: Building the Docker Image

- Prepare the Dockerfile: Ensure your Dockerfile is in the same directory as your PyQt application source code. The Dockerfile should be set up to install all necessary dependencies, including PyQt and VNC server packages.
- Build the Image:
  - Open a terminal or command prompt.
  - Navigate to the directory containing the Dockerfile.
- Run the following command to build the Docker image:

```
docker build -t my-pyqt-app
```

- `my-pyqt-app` is the default name of the Docker image. You can choose any name you prefer.

Step 2: Running the Docker Container

- Run the following command to start the container:

```
docker run -p 5900:5900 -d --name pyqt-container my-pyqt-app
```

- `-p 5900:5900` maps the VNC server port inside the container to a port on your host machine.
- `--name pyqt-container` sets the name of the container. You can use any name you prefer.
- `my-pyqt-app` is the name of the Docker image built in the previous step.

Step 3: Accessing the Application via VNC Viewer

- Find the Container's IP Address:
  - Run `docker inspect pyqt-container` to get details about the container.
  - Look for the `IPAddress` under `NetworkSettings`.
- Connect Using VNC Viewer:
  - Open VNC Viewer on your system.
  - Connect to the IP address of the container found above, followed by `:5900` (e.g., `172.17.0.2:5900`).
  - If prompted, enter the VNC password set in the Dockerfile.

Step 4: Using the Application

- Once connected through VNC Viewer, you should see the PyQt application's GUI. Interact with the application as you would normally.

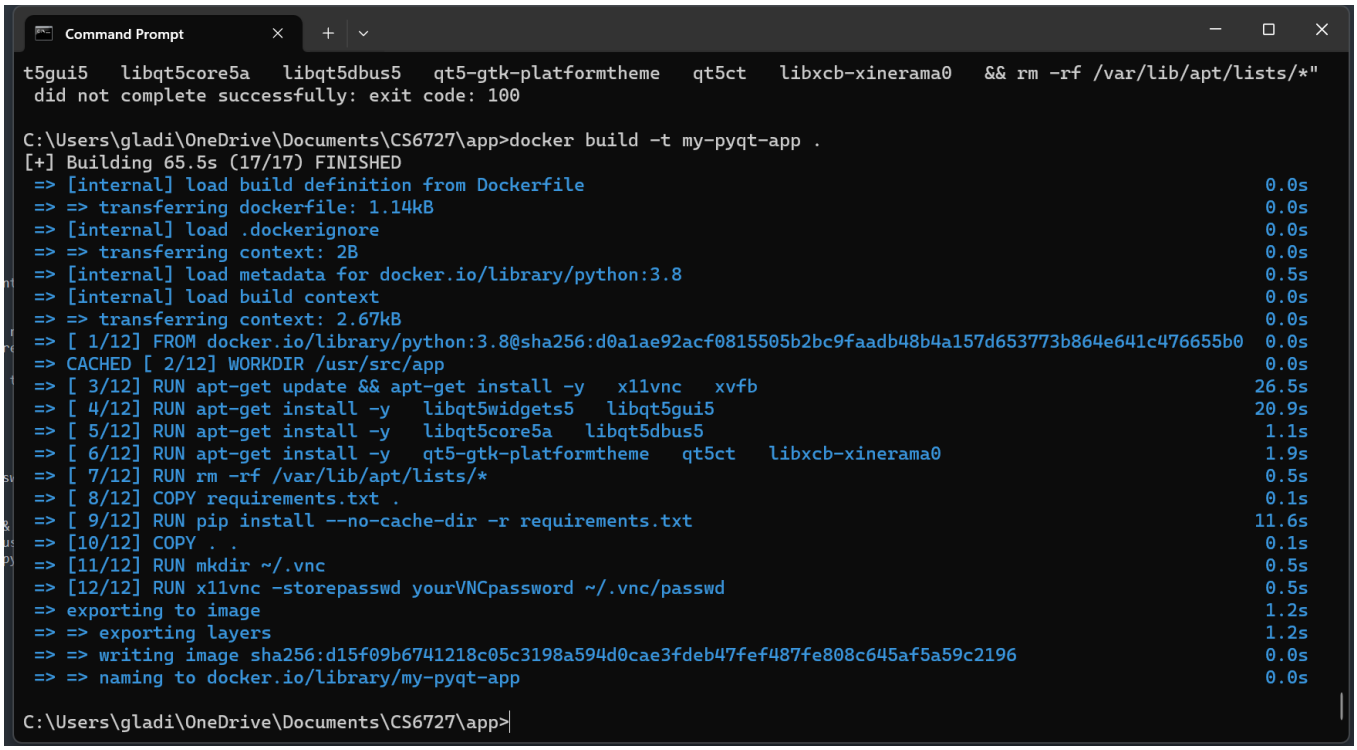
Troubleshooting

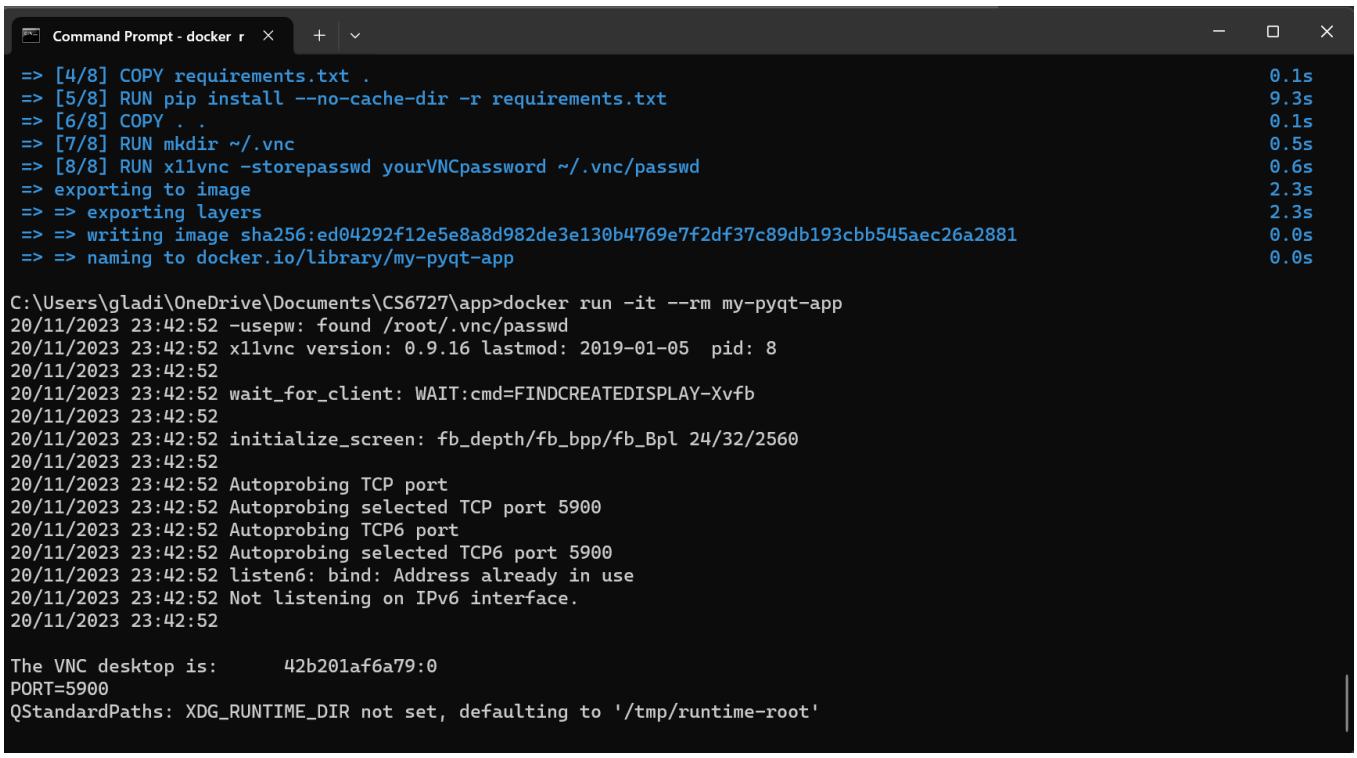
- If the application GUI does not appear, check the Docker container logs using `docker logs pyqt-container`.
- Ensure that the VNC server and Xvfb are running correctly within the container.
- If encountering network issues, verify that Docker's network settings and VNC port mappings are configured correctly.

This setup allows you to run and access PyQt GUI applications within a Docker container, leveraging the power of containerization and remote GUI access via VNC.

Example of Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:3.8

# Set the working directory in the container
WORKDIR /usr/src/app

# Install VNC, xvfb (virtual frame buffer), X11, Qt and other necessary packages for GUI
RUN apt-get update && apt-get install -y \
  x11vnc \
  xvfb

RUN apt-get install -y \
  libqt5widgets5 \
  libqt5gui5

RUN apt-get install -y \
  libqt5core5a \
  libqt5dbus5

RUN apt-get install -y \
  qt5-gtk-platformtheme \
  qt5ct \
  libxcb-xinerama0

# Clean up
RUN rm -rf /var/lib/apt/lists/*

# Copy the requirements.txt file into the container
COPY requirements.txt .

# Install Python dependencies from requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Copy the application source into the container
COPY . .

# Set up VNC (Configure as needed)
RUN mkdir ~/.vnc
# Set a VNC password here
RUN x11vnc -storepasswd yourVNCpassword ~/.vnc/passwd

# Start VNC and the application
CMD Xvfb :1 -screen 0 1024x768x16 & \
    DISPLAY=:1.0 x11vnc -forever -usepw -create & \
    DISPLAY=:1.0 python ./app1116.py
```


Command Prompt
t5gui5 libqt5core5a libqt5dbus5 qt5-gtk-platformtheme qt5ct libxcb-xinerama0 && rm -rf /var/lib/apt/lists/*"
did not complete successfully: exit code: 100
C:\Users\gladi\OneDrive\Documents\CS6727\app>docker build -t my-pyqt-app .
[+] Building 65.5s (17/17) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 1.14kB 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/python:3.8 0.5s
=> [internal] load build context 0.0s
=> => transferring context: 2.67kB 0.0s
=> [1/12] FROM docker.io/library/python:3.8@sha256:d0a1ae92acf0815505b2bc9faadb48b4a157d653773b864e641c476655b0 0.0s
=> CACHED [2/12] WORKDIR /usr/src/app 0.0s
=> [3/12] RUN apt-get update && apt-get install -y x11vnc xvfb 26.5s
=> [4/12] RUN apt-get install -y libqt5widgets5 libqt5gui5 20.9s
=> [5/12] RUN apt-get install -y libqt5core5a libqt5dbus5 1.1s
=> [6/12] RUN apt-get install -y qt5-gtk-platformtheme qt5ct libxcb-xinerama0 1.9s
=> [7/12] RUN rm -rf /var/lib/apt/lists/* 0.5s
=> [8/12] COPY requirements.txt . 0.1s
=> [9/12] RUN pip install --no-cache-dir -r requirements.txt 11.6s
=> [10/12] COPY . . 0.1s
=> [11/12] RUN mkdir ~/.vnc 0.5s
=> [12/12] RUN x11vnc -storepasswd yourVNCpassword ~/.vnc/passwd 0.5s
=> exporting to image 1.2s
=> => exporting layers 1.2s
=> => writing image sha256:d15f09b6741218c05c3198a594d0cae3fdeb47fef487fe808c645af5a59c2196 0.0s
=> => naming to docker.io/library/my-pyqt-app 0.0s
C:\Users\gladi\OneDrive\Documents\CS6727\app>


Command Prompt - docker r
=> [4/8] COPY requirements.txt . 0.1s
=> [5/8] RUN pip install --no-cache-dir -r requirements.txt 9.3s
=> [6/8] COPY . . 0.1s
=> [7/8] RUN mkdir ~/.vnc 0.5s
=> [8/8] RUN x11vnc -storepasswd yourVNCpassword ~/.vnc/passwd 0.6s
=> exporting to image 2.3s
=> => exporting layers 2.3s
=> => writing image sha256:ed04292f12e5e8a8d982de3e130b4769e7f2df37c89db193cbb545aec26a2881 0.0s
=> => naming to docker.io/library/my-pyqt-app 0.0s
C:\Users\gladi\OneDrive\Documents\CS6727\app>docker run -it --rm my-pyqt-app
20/11/2023 23:42:52 -usepw: found /root/.vnc/passwd
20/11/2023 23:42:52 x11vnc version: 0.9.16 lastmod: 2019-01-05 pid: 8
20/11/2023 23:42:52
20/11/2023 23:42:52 wait_for_client: WAIT:cmd=FINDCREATEDISPLAY-Xvfb
20/11/2023 23:42:52
20/11/2023 23:42:52 initialize_screen: fb_depth/fb_bpp/fb_Bpl 24/32/2560
20/11/2023 23:42:52
20/11/2023 23:42:52 Autoprobing TCP port
20/11/2023 23:42:52 Autoprobing selected TCP port 5900
20/11/2023 23:42:52 Autoprobing TCP6 port
20/11/2023 23:42:52 Autoprobing selected TCP6 port 5900
20/11/2023 23:42:52 listen6: bind: Address already in use
20/11/2023 23:42:52 Not listening on IPv6 interface.
20/11/2023 23:42:52
The VNC desktop is: 42b201af6a79:0
PORT=5900
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'

Appendix B: Source Codes

```
# Import necessary libraries
import sys
import boto3
import datetime
import json

from PyQt5.QtCore import Qt, QTimer, QThread, pyqtSignal
from PyQt5.QtGui import QFont, QPalette, QColor, QBrush
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout, QLabel,
                             QPushButton, QCheckBox, QGroupBox, QListWidget, QTreeWidget,
                             QTreeWidgetItem, QTextEdit, QProgressDialog, QMainWindow)

class ScanThread(QThread):
    """
    A thread class for performing AWS service scans asynchronously.

    Attributes:
        selected_services (list): A list of AWS services selected for scanning.
        progressDialog (QProgressDialog): A dialog to show the scanning progress.

    Signals:
        progressSignal (int): Signal to update progress (e.g., progress bar).
        finishedSignal (): Signal when scanning finishes.
        resultSignal (str): Signal to send scanning results.
    """
    def __init__(self, selected_services):
        """
        Initialize the scan thread with selected AWS services.

        Args:
            selected_services (list): List of selected AWS services to scan.
        """
        super().__init__()
        self._is_running = True
        self.selected_services = selected_services
        self.progressDialog = None

    progressSignal = pyqtSignal(int)  # Signal to update progress (e.g., progress bar)
    finishedSignal = pyqtSignal()     # Signal when scanning finishes
    resultSignal = pyqtSignal(str)  # Signal to send scanning results

    def run(self):
        """
        The main entry point for the thread. Scans the selected AWS services and emits signals.
        """
        service_scan_methods = {
            'EC2': self.scan_ec2,
            'S3': self.scan_s3,
            'RDS': self.scan_rds,
            'IAM': self.scan_iam,
            'Lambda': self.scan_lambda,
            'DynamoDB': self.scan_dynamodb,
            'SQS': self.scan_sqs,
            'SNS': self.scan_sns,
            'ElasticSearch': self.scan_elasticsearch,
            'Cloudfront': self.scan_cloudfront,
            'ELB': self.scan_elb,
            'Cloudwatch': self.scan_cloudwatch,
            'VPC': self.scan_vpc,
            'Secrets_Manager': self.scan_secrets_manager,
            'KMS': self.scan_kms,
            'Athena': self.scan_athena,
            'Step_functions': self.scan_step_functions,
            'Redshift': self.scan_redshift,
            'Sagemaker': self.scan_sagemaker,
            'API_gateway': self.scan_api_gateway,
            'Cognito': self.scan_cognito,
            'Route_53': self.scan_route_53,
            'WAF': self.scan_waf,
            'Inspector': self.scan_inspector,
            'GLUE': self.scan_glue,
            'EC2_Auto_scaling': self.scan_ec2_auto_scaling,
            'ECS': self.scan_ecs,
            'ECR': self.scan_ecr,
            'SWF': self.scan_swf,
            'Elastic_transcoder': self.scan_elastic_transcoder,
            'Elastic_beanstalk': self.scan_elastic_beanstalk,
            'Cloudtrail': self.scan_cloudtrail,
            'Cloudformation': self.scan_cloudformation,
            'Codedeploy': self.scan_codedeploy
        }

        for service in self.selected_services:
            scan_method = service_scan_methods.get(service)
            if scan_method:
                try:
                    scan_method()
                except Exception as e:
                    print(f"An error occurred while scanning {service}: {e}")
            else:
                print(f"No scan method found for service {service}")

        self.finishedSignal.emit()  # signal that scanning has finished

    def scan_ec2(self):
        """
        Scans EC2 instances for various vulnerabilities like public IPs, open security groups, etc.

        This method checks each EC2 instance for multiple security risks and emits signals with the findings.
        """
        self.resultSignal.emit("Starting EC2 scan...")

        ec2 = boto3.resource('ec2', region_name = 'us-east-2')
        ec2_client = boto3.client('ec2', region_name='us-east-2')
        vulnerabilities = {
            'public_ip': {'risk_level': 'High', 'recommendation': 'Remove public IPs'},
            'open_security_group': {'risk_level': 'Medium', 'recommendation': 'Narrow down security group rules'},
            # Add more vulnerabilities as needed
        }
        vulnerable_instances = []
        # Get list of all available AMIs
        latest_amis = [image['ImageId'] for image in ec2_client.describe_images(Owners=['self'])['Images']]
        for instance in ec2.instances.all():
            self.resultSignal.emit(f"Checking instance {instance.id}")

            # Check for public IPs
            if instance.public_ip_address:
                vuln_msg = (f"-> Vulnerability: Instance {instance.id} has a "
                            f"public IP address ({instance.public_ip_address}).")
                risk_msg = (f"   - Risk Level: "
                            f"{vulnerabilities['public_ip']['risk_level']}")
                rec_msg = (f"   - Recommendation: "
                           f"{vulnerabilities['public_ip']['recommendation']}")

                self.resultSignal.emit(vuln_msg)
                self.resultSignal.emit(risk_msg)
                self.resultSignal.emit(rec_msg)
                vulnerable_instances.append(instance.id)

            # Check for overly permissive rules in security groups
            for sg in instance.security_groups:
```

```
            security_group = ec2.SecurityGroup(sg['GroupId'])

            for rule in security_group.ip_permissions:
                for ip_range in rule['IpRanges']:
                    if ip_range['CidrIp'] == '0.0.0.0/0':
                        sg_msg = (f"-> Vulnerability: Security group {sg['GroupName']} "
                                  f"({sg['GroupId']}) has an overly permissive rule.")
                        self.resultSignal.emit(sg_msg)
                        vulnerable_instances.append(instance.id)
                        break  # Break out of the loop after finding a vulnerability
        # New checks
        if not instance.key_pair:
            self.resultSignal.emit(f"-> Vulnerability: Instance {instance.id} does not have a key pair.")
            vulnerable_instances.append(instance.id)

        if not instance.iam_instance_profile:
            self.resultSignal.emit(f"-> Vulnerability: Instance {instance.id} does not have an IAM role.")
            vulnerable_instances.append(instance.id)

        # Check if attached volumes are encrypted
        for volume in instance.volumes.all():
            if not volume.encrypted:
                volume_msg = (f"-> Vulnerability: Volume {volume.id} attached to "
                              f"instance {instance.id} is not encrypted.")
                self.resultSignal.emit(volume_msg)
                vulnerable_instances.append(instance.id)

        # Check if instance is in default VPC
        if instance.vpc.is_default:
            self.resultSignal.emit(f"-> Vulnerability: Instance {instance.id} is in the default VPC.")
            vulnerable_instances.append(instance.id)

        # Check if monitoring is disabled
        if instance.monitoring['State'] != 'enabled':
            self.resultSignal.emit(f"-> Vulnerability: Monitoring is disabled for instance {instance.id}.")
            vulnerable_instances.append(instance.id)

        # Check for unrestricted ICMP access
        for sg in instance.security_groups:
            security_group = ec2.SecurityGroup(sg['GroupId'])
            for rule in security_group.ip_permissions:
                for ip_range in rule['IpRanges']:
                    if ip_range['CidrIp'] == '0.0.0.0/0' and 'icmp' in rule['IpProtocol']:
                        icmp_msg = (f"-> Vulnerability: Security group {sg['GroupName']} "
                                    f"({sg['GroupId']}) allows unrestricted ICMP.")
                        self.resultSignal.emit(icmp_msg)
                        vulnerable_instances.append(instance.id)
                        break
        iam_client = boto3.client('iam')
        try:
            root_account_last_used = iam_client.get_account_summary()['SummaryMap']['AccountLastUsed']
            if root_account_last_used:
                self.resultSignal.emit(f"-> Vulnerability: The root account was used recently on {root_account_last_used}.")
                vulnerable_instances.append(instance.id)
        except ClientError:
            pass

        # Check for sensitive data in user data
        try:
            user_data_resp = ec2_client.describe_instance_attribute(
                Attribute='userData', InstanceId=instance.id)
            user_data = user_data_resp['UserData']

            if user_data:
                sensitive_keywords = ['password', 'secret', 'token']
                for keyword in sensitive_keywords:
                    if keyword in user_data['Value']:
                        sensitive_msg = (f"-> Vulnerability: Sensitive data found in user data "
                                         f"of instance {instance.id}.")
                        self.resultSignal.emit(sensitive_msg)
                        vulnerable_instances.append(instance.id)
                        break
        except ClientError:
            pass

        # Check for usage of old AMIs
        if instance.image_id not in latest_amis:
            self.resultSignal.emit(f"-> Vulnerability: Instance {instance.id} is using an old AMI {instance.image_id}.")
            vulnerable_instances.append(instance.id)

    if vulnerable_instances:
        self.resultSignal.emit(f"Vulnerable instances found: {', '.join(vulnerable_instances)}")
    else:
        self.resultSignal.emit("No vulnerable instances found.")

    checks_done = 0
    total_checks = 10  # You can adjust this based on how many checks you have

    for instance in ec2.instances.all():

        #... [After each check in your scanning logic]
        checks_done += 1
        progress = (checks_done / total_checks) * 100
        self.progressSignal.emit(progress)
        if not self._is_running:
            return

def scan_s3(self):
    """
    Scans Amazon S3 buckets for various security vulnerabilities.

    This function checks each S3 bucket for issues such as public access through ACLs and bucket policies,
    lack of versioning, MFA Delete not being enabled, logging not being enabled, lack of default encryption,
    public bucket policies, and exposed CORS configurations. It emits signals to report the findings.

    Each vulnerability check attempts to retrieve specific bucket configurations and assesses them
    against security best practices. Found vulnerabilities are collected and reported.

    Note:
    - This function relies on the boto3 AWS SDK for Python to interact with Amazon S3.
    - It handles exceptions to ensure the scan continues even if some checks fail.

    Emits:
    - resultSignal: Signals the start of the scan, the checking of each bucket, and the results of the scan.
    - resultSignal with formatted vulnerability messages for each identified vulnerability.
    """
    self.resultSignal.emit("Starting S3 scan...")

    s3_client = boto3.client('s3')

    # Holding all vulnerabilities
    vulnerable_buckets_info = {}

    response = s3_client.list_buckets()

    # Print all buckets to debug
    print("All buckets:", [bucket['Name'] for bucket in response['Buckets']])

    for bucket in response['Buckets']:
        bucket_name = bucket['Name']
        self.resultSignal.emit(f"Checking bucket {bucket_name}")
```

```
vulnerabilities = []

# Public Access through ACL
try:
    bucket_acl = s3_client.get_bucket_acl(Bucket=bucket_name)
    for grant in bucket_acl['Grants']:
        if grant['Grantee'].get('Type') == 'Group' and 'AllUsers' in grant['Grantee'].get('URI', ''):
            vulnerabilities.append({
                "risk": "High",
                "CVE": "CVE-2021-32717",
                "CVSS": "7.5",
                "Custom Risk Rating": "High",
                "message": "Bucket has public access through ACL."
            })
            break
except ClientError:
    self.resultSignal.emit(f"Could not retrieve ACL for bucket {bucket_name}. Skipping.")

# Public Access through Bucket Policy
try:
    bucket_policy = s3_client.get_bucket_policy(Bucket=bucket_name)
    policy_string = bucket_policy['Policy']
    policy_json = json.loads(policy_string)

    for statement in policy_json.get('Statement', []):
        is_effect_allow = statement.get('Effect') == 'Allow'
        is_principal_star = statement.get('Principal') == '*'

        if is_effect_allow and is_principal_star:
            vulnerabilities.append({
                "risk": "High",
                "CVE": "CVE-2021-32717",  # Adjust CVE according to the specific vulnerability
                "CVSS": "7.5",  # Adjust CVSS score
                "Custom Risk Rating": "High",
                "message": "Bucket has public access through Policy."
            })
            break

except ClientError as e:
    if e.response['Error']['Code'] != 'NoSuchBucketPolicy':
        self.resultSignal.emit(f"Could not retrieve bucket policy for bucket {bucket_name}. Skipping.")

# Check for versioning status in S3 bucket
try:
    versioning_status = s3_client.get_bucket_versioning(
        Bucket=bucket_name)

    if versioning_status.get('Status') != 'Enabled':
        vulnerabilities.append({
            "risk": "Medium",
            "message": ("Bucket does not have versioning enabled. "
                        "Enable versioning to protect against unintended "
                        "changes or deletions.")
        })

except ClientError:
    versioning_error_msg = (f"Could not retrieve versioning status for "
                            f"bucket {bucket_name}. Skipping.")
    self.resultSignal.emit(versioning_error_msg)
# Print ACL
try:
    bucket_acl = s3_client.get_bucket_acl(Bucket=bucket_name)
    print(f"ACL for {bucket_name}: {json.dumps(bucket_acl, indent=2)}")
except ClientError:
    self.resultSignal.emit(f"Could not retrieve ACL for bucket {bucket_name}. Skipping.")

# Print Bucket Policy
try:
    bucket_policy = s3_client.get_bucket_policy(Bucket=bucket_name)
    policy_string = bucket_policy['Policy']
    print(f"Policy for {bucket_name}: {policy_string}")
except ClientError as e:
    if e.response['Error']['Code'] != 'NoSuchBucketPolicy':
        self.resultSignal.emit(f"Could not retrieve bucket policy for bucket {bucket_name}. Skipping.")
# MFA Delete check for S3 bucket
try:
    versioning_status = s3_client.get_bucket_versioning(
        Bucket=bucket_name)
    mfa_delete = versioning_status.get('MFADelete', 'Disabled')

    if mfa_delete == 'Disabled':
        vulnerabilities.append({
            "risk": "Medium",
            "message": ("Bucket does not have MFA Delete enabled. "
                        "Enable MFA Delete to add an extra layer of "
                        "security.")
        })

except ClientError:
    mfa_delete_error_msg = (f"Could not retrieve MFA Delete status for "
                            f"bucket {bucket_name}. Skipping.")
    self.resultSignal.emit(mfa_delete_error_msg)

# Logging
try:
    logging_status = s3_client.get_bucket_logging(Bucket=bucket_name)
    if not logging_status.get('LoggingEnabled'):
        vulnerabilities.append({
            "risk": "Low",
            "message": "Bucket does not have logging enabled. Enable logging for auditing and monitoring."
        })
except ClientError:
    self.resultSignal.emit(f"Could not retrieve logging status for bucket {bucket_name}. Skipping.")

# Default Encryption
try:
    encryption_status = s3_client.get_bucket_encryption(Bucket=bucket_name)
    if not encryption_status.get('ServerSideEncryptionConfiguration'):
        vulnerabilities.append({
            "risk": "Medium",
            "message": "Bucket does not have default encryption enabled. \
            Enable default encryption to secure your data."
        })
except ClientError:
    vulnerabilities.append({
        "risk": "Medium",
        "message": "Bucket does not have default encryption enabled. \
        Enable default encryption to secure your data."
    })
# Public Bucket Policies
try:
    policy = s3_client.get_bucket_policy(Bucket=bucket_name)
    if 'Public' in policy.get('Policy', ''):
        vulnerabilities.append({"risk": "High", "message": "Bucket has a public bucket policy. \
        Modify the bucket policy to restrict public access."})
except ClientError as e:
    if e.response['Error']['Code'] != 'NoSuchBucketPolicy':
        self.resultSignal.emit(f"Could not retrieve bucket policy for bucket {bucket_name}. Skipping.")

# Exposed CORS Configurations
try:
    cors_status = s3_client.get_bucket_cors(Bucket=bucket_name)
```

```
            for rule in cors_status.get('CORSRules', []):
                if '*' in rule.get('AllowedOrigins', []):
                    vulnerabilities.append({"risk": "Medium", "message": "Bucket has exposed CORS configurations. \
                    Tighten CORS rules to restrict origins."})
                    break
        except ClientError as e:
            if e.response['Error']['Code'] != 'NoSuchCORSConfiguration':
                self.resultSignal.emit(f"Could not retrieve CORS status for bucket {bucket_name}. Skipping.")

        # If any vulnerabilities are found, store them
        if vulnerabilities:
            vulnerable_buckets_info[bucket_name] = vulnerabilities
    if vulnerable_buckets_info:
        for bucket, vulns in vulnerable_buckets_info.items():
            self.resultSignal.emit(f"Bucket: {bucket}")
            for vuln in vulns:
                message = f"  - Risk: {vuln['risk']}, Message: {vuln['message']}"
                if 'CVSS' in vuln:
                    message += f", CVSS: {vuln['CVSS']}"
                if 'CVE' in vuln:
                    message += f", CVE: {vuln['CVE']}"
                if 'Custom Risk Rating' in vuln:
                    message += f", Custom Risk Rating: {vuln['Custom Risk Rating']}"
                self.resultSignal.emit(message)
    else:
        self.resultSignal.emit("No vulnerable buckets found.")

def scan_rds(self):
    """
    Scans Amazon RDS instances for various security vulnerabilities.

    This function iterates through all RDS instances in a specified AWS region and checks for
    several security-related configurations. It checks for public accessibility, whether Multi-AZ
    is disabled, logging configuration, encryption at rest, IAM Database Authentication,
    automated backups and their retention periods, and deletion protection.

    Vulnerabilities are reported for each instance where the configuration does not align with
    best security practices. The function emits signals with the findings for each instance and
    a summary at the end.

    Note:
    - This function uses the boto3 library to interact with the AWS RDS service.
    - It captures exceptions to handle cases where RDS instances cannot be retrieved or scanned.

    Emits:
    - resultSignal: Signals the start of the RDS scan, the checking of each instance, and the results of the scan.
    - resultSignal with formatted vulnerability messages for each identified vulnerability.
    - resultSignal with a summary of all vulnerable RDS instances found or a message if none are found.
    """
    self.resultSignal.emit("Starting rds scan...")
    try:
        # Initialize a session using Amazon RDS
        rds_client = boto3.client('rds', region_name='us-west-2')

        # Create a list to hold any RDS instances we consider to be 'vulnerable'
        vulnerable_instances = []

        # Retrieve a list of all RDS instances
        response = rds_client.describe_db_instances()

        for instance in response['DBInstances']:
            db_identifier = instance['DBInstanceIdentifier']
            self.resultSignal.emit(f"Checking RDS instance {db_identifier}")

            # Check for public accessibility
            if instance['PubliclyAccessible']:
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} is publicly accessible.")
                vulnerable_instances.append(db_identifier)

            # Check if Multi-AZ is disabled
            if not instance['MultiAZ']:
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} \
                does not have Multi-AZ deployments enabled.")
                vulnerable_instances.append(db_identifier)

            # Check for logging
            if not instance.get('EnabledCloudwatchLogsExports'):
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} does not have logging enabled.")
                vulnerable_instances.append(db_identifier)

            # Check for encryption
            if not instance.get('StorageEncrypted'):
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} \
                does not have encryption at rest enabled.")
                vulnerable_instances.append(db_identifier)

            # Check for IAM Database Authentication
            if not instance.get('IAMDatabaseAuthenticationEnabled'):
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} \
                does not have IAM database authentication enabled.")
                vulnerable_instances.append(db_identifier)

            # Check for automated backups and retention period
            if not instance['BackupRetentionPeriod']:
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} \
                does not have automated backups enabled.")
                vulnerable_instances.append(db_identifier)
            elif instance['BackupRetentionPeriod'] < 7:
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} \
                has a backup retention period less than 7 days.")
                vulnerable_instances.append(db_identifier)

            # Check for deletion protection
            if not instance.get('DeletionProtection'):
                self.resultSignal.emit(f"-> Vulnerability: RDS instance {db_identifier} \
                does not have deletion protection enabled.")
                vulnerable_instances.append(db_identifier)
                # This part should be outside the for loop to provide a summary after all instances have been checked

        if vulnerable_instances:
            self.resultSignal.emit(f"Vulnerable RDS instances found: {', '.join(vulnerable_instances)}")
        else:
            self.resultSignal.emit("No vulnerable RDS instances found.")

    except ClientError as e:
        self.resultSignal.emit(f"Could not retrieve RDS instances due to {e}. Skipping...")
def scan_iam(self):
    """
    Scans AWS Identity and Access Management (IAM) for various security vulnerabilities.

    This function performs multiple checks on the IAM configuration of the AWS account, including:
    - Whether the root account has Multi-Factor Authentication (MFA) enabled.
    - Compliance of IAM users with the account password policy.
    - Presence of console access for IAM users.
    - Roles with trust relationships to entities outside the AWS account.
    - IAM users with active SSH keys.
    - IAM users without assigned credentials.
    - Unused IAM roles.
    - Roles with policies lacking conditions.
    - IAM users with administrative privileges.
    - Old access keys.
    - IAM users without MFA enabled.
```

```
Each check aims to identify potential security risks in the IAM setup, and findings are reported through emitted signals.
The method handles various scenarios and potential exceptions to ensure comprehensive coverage of IAM configurations.

Note:
- The method uses boto3 for interactions with the AWS IAM service.
- It handles ClientError exceptions for robust error management during the scan.

Emits:
- resultSignal: Signals the start of the IAM scan and the results, including identified vulnerabilities.
- resultSignal with detailed messages for each identified vulnerability.
- resultSignal with a summary of vulnerable IAM users and roles found, or a message if none are found.
"""
self.resultSignal.emit("Starting IAM scan...")

iam_client = boto3.client('iam')
vulnerable_users = []
vulnerable_roles = []

try:
    # 1. Check if the root user has MFA enabled
    account_mfa = iam_client.get_account_summary()['SummaryMap']['AccountMFAEnabled']
    if not account_mfa:
        self.resultSignal.emit("-> Vulnerability: Root account does not have MFA enabled.")

    # 2. Check for IAM users with password policy violations
    password_policy = iam_client.get_account_password_policy()['PasswordPolicy']
    if not password_policy.get('MinimumPasswordLength', 0) >= 14 or \
       not password_policy.get('RequireSymbols') or \
       not password_policy.get('RequireUppercaseCharacters') or \
       not password_policy.get('RequireLowercaseCharacters') or \
       not password_policy.get('RequireNumbers'):
        self.resultSignal.emit("-> Vulnerability: IAM users might be violating password policy.")

    # 3. Check for IAM users with console passwords
    users = iam_client.list_users()['Users']
    for user in users:
        login_profile_exists = True
        try:
            iam_client.get_login_profile(UserName=user['UserName'])
        except ClientError as e:
            if e.response['Error']['Code'] == 'NoSuchEntity':
                login_profile_exists = False
        if login_profile_exists:
            self.resultSignal.emit(f"-> Vulnerability: IAM user {user['UserName']} has console access.")
            vulnerable_users.append(user['UserName'])

    # 4. Check roles with trusted entities outside of your account
    roles = iam_client.list_roles()['Roles']
    for role in roles:
        trust_relationship = json.loads(role['AssumeRolePolicyDocument'])
        for statement in trust_relationship.get('Statement', []):
            if statement.get('Effect') == 'Allow' and 'AWS' in statement.get('Principal', {}):
                principal_aws = statement['Principal']['AWS']
                if not principal_aws.startswith(f"arn:aws:iam::{role['Arn'].split(':')[4]}:"):
                    self.resultSignal.emit(f"-> Vulnerability: IAM role {role['RoleName']} \
                    trusts an entity outside of your AWS account.")
                    vulnerable_roles.append(role['RoleName'])

    # 5. Check for IAM users with SSH keys
    users = iam_client.list_users()['Users']
    for user in users:
        ssh_keys = iam_client.list_ssh_public_keys(UserName=user['UserName'])['SSHPublicKeys']
        if [key for key in ssh_keys if key['Status'] == 'Active']:
            self.resultSignal.emit(f"-> Vulnerability: IAM user {user['UserName']} has active SSH keys.")
            vulnerable_users.append(user['UserName'])

    # 6. Check for users without credentials
    for user in users:
        passwords = iam_client.list_user_policies(UserName=user['UserName'])['PolicyNames']
        access_keys = iam_client.list_access_keys(UserName=user['UserName'])['AccessKeyMetadata']
        if not passwords and not [key for key in access_keys if key['Status'] == 'Active']:
            self.resultSignal.emit(f"-> Vulnerability: IAM user {user['UserName']} \
            doesn't have assigned credentials.")
            vulnerable_users.append(user['UserName'])

    # 7. Check for unused IAM roles
    roles = iam_client.list_roles()['Roles']
    for role in roles:
        last_used = role.get('RoleLastUsed')
        if last_used:
            if 'LastUsedDate' not in last_used or last_used['LastUsedDate'] < datetime.datetime.now(datetime.timezone.utc) - datetime.timedelta(days=90):
                self.resultSignal.emit(f"-> Vulnerability: IAM role {role['RoleName']} \
                has not been used in the last 90 days.")
                vulnerable_roles.append(role['RoleName'])

    # 8. Service-specific roles without conditions
    for role in roles:
        policies = iam_client.list_role_policies(RoleName=role['RoleName'])['PolicyNames']
        for policy_name in policies:
            policy = iam_client.get_role_policy(RoleName=role['RoleName'], PolicyName=policy_name)
            if 'Condition' not in policy['PolicyDocument']:
                self.resultSignal.emit(f"-> Vulnerability: IAM role {role['RoleName']} \
                has a policy without conditions.")
                vulnerable_roles.append(role['RoleName'])

    # 9. Check for IAM users with admin privileges
    groups_response = iam_client.list_groups()
    for group in groups_response['Groups']:
        group_name = group['GroupName']
        group_policies = iam_client.list_group_policies(GroupName=group_name)['PolicyNames']
        attached_policies = iam_client.list_attached_group_policies(GroupName=group_name)['AttachedPolicies']

        if 'AdministratorAccess' in group_policies or any(p['PolicyName'] == 'AdministratorAccess' for p in attached_policies):
            group_users = iam_client.get_group(GroupName=group_name)['Users']
            for user in group_users:
                username = user['UserName']
                self.resultSignal.emit(f"-> Vulnerability: IAM user {username} has admin privileges.")
                vulnerable_users.append(username)

    # 10. Check for old access keys
    for user in users:
        username = user['UserName']
        access_keys = iam_client.list_access_keys(UserName=username)['AccessKeyMetadata']
        for key in access_keys:
            if key['Status'] == 'Active':
                last_used_response = iam_client.get_access_key_last_used(AccessKeyId=key['AccessKeyId'])

                if 'LastUsedDate' in last_used_response:
                    last_used_date = last_used_response['LastUsedDate']
                    if (datetime.datetime.now(last_used_date.tzinfo) - last_used_date).days > 90:
                        self.resultSignal.emit(f"-> Vulnerability: IAM user {username} \
                        has an active access key not used for 90 days.")
                        vulnerable_users.append(username)

    # 11. Check for MFA (Mock)
    for user in users:
        mfa_devices = iam_client.list_mfa_devices(UserName=user['UserName'])
        if not mfa_devices['MFADevices']:
            self.resultSignal.emit(f"-> Vulnerability: IAM user {user['UserName']} does not have MFA enabled.")
            vulnerable_users.append(user['UserName'])
```

```
        except ClientError as e:
            self.resultSignal.emit(f"Could not retrieve IAM information: {e}")

        if vulnerable_users:
            self.resultSignal.emit(f"Vulnerable IAM users found: {vulnerable_users}")
        else:
            self.resultSignal.emit("No vulnerable IAM users found.")

        if vulnerable_roles:
            self.resultSignal.emit(f"Vulnerable IAM roles found: {vulnerable_roles}")
        else:
            self.resultSignal.emit("No vulnerable IAM roles found.")

    def scan_lambda(self):
            self.resultSignal.emit("Starting Lambda scan...")
            lambda_client = boto3.client('lambda', region_name='us-west-1')
            vulnerable_lambdas = []
            """
            Scans AWS Lambda functions for security vulnerabilities related to invocation permissions.

            This function checks each Lambda function in a specified AWS region for open invocation permissions,
            which could potentially allow unauthorized access. It identifies functions where the associated
            resource-based policy has 'Allow' statements with a wildcard ('*') principal, indicating public accessibility.

            The scan involves listing all Lambda functions using a paginator and retrieving their policies.
            Vulnerabilities are reported for each function that meets the criteria.

            Note:
            - The function uses boto3 to interact with the AWS Lambda service and handles paginated responses for \
            large numbers of functions.
            - It captures both ClientError for AWS SDK errors and generic exceptions for robust error handling.

            Emits:
            - resultSignal: Signals the start of the Lambda scan, the results for each function, and a summary of the findings.
            - resultSignal with detailed messages for each identified vulnerability.
            - resultSignal with a summary of all vulnerable Lambda functions found or a message if none are found.

            Exceptions:
            - ClientError: If an AWS SDK error occurs while retrieving Lambda information.
            - Exception: For any other unexpected errors during the scan.
            """
            try:
                # List Lambda functions
                paginator = lambda_client.get_paginator('list_functions')
                iterator = paginator.paginate()

                for page in iterator:
                    for function in page['Functions']:
                        policy_str = lambda_client.get_policy(FunctionName=function['FunctionName'])['Policy']
                        policy = json.loads(policy_str)

                        if "Statement" in policy:
                            for statement in policy['Statement']:
                                if ('Effect' in statement and statement['Effect'] == 'Allow'
                                        and 'Principal' in statement and statement['Principal'] == '*'):
                                    vulnerable_lambdas.append(function['FunctionName'])
                                    self.resultSignal.emit(f"-> Vulnerability: Lambda function {function['FunctionName']}\
                                    has open invocation permissions.")

            except ClientError as e:
                self.resultSignal.emit(f"Could not retrieve Lambda information due to client error: {e}")
            except Exception as e:
                self.resultSignal.emit(f"An unexpected error occurred: {e}")

            if vulnerable_lambdas:
                self.resultSignal.emit(f"Vulnerable Lambda functions found: {', '.join(vulnerable_lambdas)}")
            else:
                self.resultSignal.emit("No vulnerable Lambda functions found.")

    def scan_dynamodb(self):
        """
        Scans DynamoDB tables in a specific AWS region for server-side encryption vulnerabilities.

        This function iterates over all DynamoDB tables and checks if server-side encryption (SSE)
        is enabled. It flags tables where SSE is not enabled as vulnerable, considering server-side
        encryption as a security best practice for protecting data at rest.

        The scan involves listing all DynamoDB tables using the boto3 DynamoDB client and then
        describing each table to check the status of its SSE configuration. Tables without enabled
        SSE are reported as vulnerabilities.

        Note:
        - The function uses boto3 to interact with the AWS DynamoDB service.
        - It works in the 'us-east-2' region, which should be adjusted if necessary.

        Emits:
        - resultSignal: Signals the start of the DynamoDB scan and reports findings for each table.
        - resultSignal with messages for each table where server-side encryption is not enabled.
        """
        self.resultSignal.emit("Starting DynamoDB scan...")
        dynamodb = boto3.client('dynamodb', region_name = 'us-east-2')

        # List tables
        tables = dynamodb.list_tables()
        for table_name in tables['TableNames']:
            # Describe table to check for encryption
            table_description = dynamodb.describe_table(TableName=table_name)
            sse_description = table_description['Table'].get('SSEDescription', {})
            if sse_description.get('Status') != 'ENABLED':
                self.resultSignal.emit(f"-> Vulnerability: DynamoDB table {table_name} \
                has server-side encryption disabled.")
    def scan_sqs(self):
        """
        Scans Amazon Simple Queue Service (SQS) for queues with potential security vulnerabilities.

        This function checks each SQS queue in the specified AWS region for public access vulnerabilities.
        It identifies queues where the associated access policy allows unrestricted ('*') access, indicating
        that the queue could be publicly accessible and therefore a potential security risk.

        The scan involves listing all SQS queues and retrieving their policies. Queues with policies that
        grant 'Allow' permission to a wildcard principal ('*') are reported as vulnerabilities.

        Note:
        - The function uses boto3 to interact with the AWS SQS service.
        - It operates in the 'us-east-2' region, which should be configured as required.

        Emits:
        - resultSignal: Signals the start of the SQS scan, and reports findings for each queue.
        - resultSignal with messages for each queue identified as having public access vulnerabilities.
        """
        self.resultSignal.emit("Starting SQS scan...")
        sqs = boto3.client('sqs', region_name='us-east-2')

        # List queues
        queues = sqs.list_queues()
        for queue_url in queues['QueueUrls']:
            attributes = sqs.get_queue_attributes(
                QueueUrl=queue_url,
                AttributeNames=['Policy']
            )
            if 'Policy' in attributes['Attributes']:
                policy = json.loads(attributes['Attributes']['Policy'])
```

```
            for statement in policy.get('Statement', []):
                principal = statement.get('Principal', {})

                if isinstance(principal, dict):
                    aws_principal = principal.get('AWS')
                else:  # Principal is a string, not a dict
                    aws_principal = principal

                if statement.get('Effect') == 'Allow' and aws_principal == '*':
                    self.resultSignal.emit(f"-> Vulnerability: SQS queue {queue_url} has public access.")
    def scan_sns(self):
        """
        Scans Amazon Simple Notification Service (SNS) topics for security vulnerabilities.

        This function checks each SNS topic in the specified AWS region for public access vulnerabilities.
        It identifies topics where the associated access policy allows unrestricted ('*') access, indicating
        that the topic could be publicly accessible and therefore a potential security risk.

        The scan involves listing all SNS topics and retrieving their policies. Topics with policies that
        grant 'Allow' permission to a wildcard principal ('*') are reported as vulnerabilities.

        Note:
        - The function uses boto3 to interact with the AWS SNS service.
        - It operates in the 'us-east-2' region, which should be configured as required.

        Emits:
        - resultSignal: Signals the start of the SNS scan, and reports findings for each topic.
        - resultSignal with messages for each topic identified as having public access vulnerabilities.
        """
        self.resultSignal.emit("Starting SNS scan...")
        sns = boto3.client('sns', region_name='us-east-2')

        # List topics
        topics = sns.list_topics()
        for topic in topics['Topics']:
            topic_arn = topic['TopicArn']  # Corrected this line
            attributes = sns.get_topic_attributes(TopicArn=topic_arn)
            policy = json.loads(attributes['Attributes']['Policy'])
            for statement in policy.get('Statement', []):
                principal = statement.get('Principal', {})
                if isinstance(principal, dict):
                    aws_principal = principal.get('AWS')
                else:  # Principal is a string, not a dict
                    aws_principal = principal
                if statement.get('Effect') == 'Allow' and aws_principal == '*':
                    self.resultSignal.emit(f"-> Vulnerability: SNS topic {topic_arn} has public access.")
    def scan_elasticsearch(self):
        """
        Scans AWS Elasticsearch domains for public access vulnerabilities.

        This function iterates over all Elasticsearch domains in the specified AWS region and
        checks if their access policies allow unrestricted public access. A domain is flagged as
        vulnerable if its access policy includes 'Allow' statements with a wildcard ('*') principal,
        indicating that it could be publicly accessible.

        The scan involves listing all Elasticsearch domains and then retrieving and parsing their
        access policies. Domains with policies that potentially allow public access are reported as
        vulnerabilities.

        Note:
        - The function uses boto3 to interact with the AWS Elasticsearch service.
        - It operates in the 'us-east-2' region, which should be configured as required.

        Emits:
        - resultSignal: Signals the start of the Elasticsearch scan and reports findings for each domain.
        - resultSignal with messages for each domain identified as having public access vulnerabilities.
        """
        self.resultSignal.emit("Starting ElasticSearch scan...")
        es = boto3.client('es', region_name = 'us-east-2')

        # List domains
        domains = es.list_domain_names()
        for domain in domains['DomainNames']:
            domain_name = domain['DomainName']
            domain_info = es.describe_elasticsearch_domain(DomainName=domain_name)
            access_policies = json.loads(domain_info['DomainStatus'].get('AccessPolicies', '{}'))
            for statement in access_policies.get('Statement', []):
                if statement.get('Effect') == 'Allow' and statement.get('Principal', {}).get('AWS') == '*':
                    self.resultSignal.emit(f"-> Vulnerability: ElasticSearch domain {domain_name} is publicly accessible.")
    def scan_cloudfront(self):
        """
        Scans AWS CloudFront distributions for specific configuration vulnerabilities.

        This function checks each CloudFront distribution for certain vulnerabilities, such as the lack
        of a default root object. The absence of a default root object in a CloudFront distribution is
        considered a vulnerability as it could lead to unintended behavior or expose sensitive files.

        The scan involves listing all CloudFront distributions and checking their configurations against
        defined vulnerability criteria. Identified vulnerabilities, along with their risk levels and
        recommendations, are reported.

        Note:
        - The function uses boto3 to interact with the AWS CloudFront service.
        - Vulnerabilities and their respective recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the CloudFront scan, reports the status of each distribution,
          and provides a summary of all vulnerable distributions found or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability.
        """
        self.resultSignal.emit("Starting CloudFront scan...")
        cloudfront_client = boto3.client('cloudfront')
        distributions = cloudfront_client.list_distributions()['DistributionList']['Items']

        vulnerabilities = {
            'default_root_object': {'risk_level': 'Low', 'recommendation': 'Set a default root object'},
        }
        vulnerable_distributions = []

        for distribution in distributions:
            self.resultSignal.emit(f"Checking distribution {distribution['Id']}")
            if not distribution.get('DefaultRootObject'):
                self.resultSignal.emit(f"-> Vulnerability: Distribution {distribution['Id']} \
                doesn't have a default root object.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['default_root_object']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['default_root_object']['recommendation']}")
                vulnerable_distributions.append(distribution['Id'])

        # If vulnerable distributions are found, emit a single formatted string message
        if vulnerable_distributions:
            distributions_string = ', '.join(vulnerable_distributions)
            message = f"Vulnerable distributions found: {distributions_string}"
            self.resultSignal.emit(message)
        else:
            self.resultSignal.emit("No vulnerable distributions found.")
    def scan_elb(self):
        """
        Scans AWS Elastic Load Balancers (ELB) for security vulnerabilities, specifically focusing on the use of HTTP listeners.

        This function iterates over all ELBs in the AWS account and checks for listeners that use HTTP protocol instead of HTTPS.
        The presence of HTTP listeners is flagged as a vulnerability due to the lack of encryption, which could expose sensitive data.
```

```
        The scan involves listing all ELBs, retrieving their listener configurations, and checking for any listeners using HTTP.
        Identified vulnerabilities are reported, including the risk level and a recommendation to switch to HTTPS.

        Note:
        - The function uses boto3 to interact with the AWS ELB service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the ELB scan and reports findings for each load balancer.
        - resultSignal with messages for each load balancer identified as having HTTP listeners and the respective recommendation.
        """
        self.resultSignal.emit("Starting ELB scan...")
        elb_client = boto3.client('elbv2')
        load_balancers = elb_client.describe_load_balancers()['LoadBalancers']

        vulnerabilities = {
            'http_listener': {'risk_level': 'Medium', 'recommendation': 'Use HTTPS instead of HTTP'},
        }
        vulnerable_lbs = []

        for lb in load_balancers:
            self.resultSignal.emit(f"Checking load balancer {lb['LoadBalancerName']}")
            listeners = elb_client.describe_listeners(LoadBalancerArn=lb['LoadBalancerArn'])['Listeners']
            for listener in listeners:
                if listener['Protocol'] == 'HTTP':
                    self.resultSignal.emit(f"-> Vulnerability: Load balancer {lb['LoadBalancerName']} uses HTTP listener.")
                    self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['http_listener']['risk_level']}")
                    self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['http_listener']['recommendation']}")
                    vulnerable_lbs.append(lb['LoadBalancerName'])
                    break

        if vulnerable_lbs:
            self.resultSignal.emit("Vulnerable load balancers found:", vulnerable_lbs)
        else:
            self.resultSignal.emit("No vulnerable load balancers found.")
    def scan_cloudwatch(self):
        """
        Scans AWS CloudWatch for missing alarms on metrics, indicating potential monitoring gaps.

        This function checks the AWS CloudWatch service for the presence of metric alarms.
        The absence of any alarms is flagged as a vulnerability, as it suggests a lack of monitoring
        that could lead to missed critical alerts or performance issues.

        The scan involves querying the CloudWatch service for all defined metric alarms.
        If no alarms are found, it is reported as a vulnerability, along with a risk level and a
        recommendation to set up necessary alarms.

        Note:
        - The function uses boto3 to interact with the AWS CloudWatch service.
        - The risk levels and recommendations for vulnerabilities are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the CloudWatch scan and reports the findings.
        - resultSignal with a message if no CloudWatch alarms are set, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting CloudWatch scan...")
        cloudwatch_client = boto3.client('cloudwatch')
        alarms = cloudwatch_client.describe_alarms()['MetricAlarms']

        vulnerabilities = {
            'no_alarm': {'risk_level': 'Medium', 'recommendation': 'Set up an alarm'},
        }
        vulnerable_metrics = []

        if not alarms:
            self.resultSignal.emit("-> Vulnerability: No CloudWatch Alarms are set.")
            self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['no_alarm']['risk_level']}")
            self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['no_alarm']['recommendation']}")
            vulnerable_metrics.append('All metrics')

        if vulnerable_metrics:
            self.resultSignal.emit("Vulnerable metrics found:", vulnerable_metrics)
        else:
            self.resultSignal.emit("No vulnerable metrics found.")
    def scan_vpc(self):
        """
        Scans AWS Virtual Private Clouds (VPCs) for specific configuration vulnerabilities, particularly focusing on default VPCs.

        This function checks each VPC in the AWS account to determine if it is a default VPC, which is considered a vulnerability.
        Using default VPCs can be a risk due to their common, well-known configurations and potential for misconfiguration.

        The scan involves listing all VPCs and checking their 'IsDefault' attribute. VPCs that are marked as default are reported
        as vulnerabilities, along with a risk level and a recommendation to consider using non-default VPCs.

        Note:
        - The function uses boto3 to interact with the AWS EC2 service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the VPC scan, reports the status of each VPC, and provides a summary of all vulnerable VPCs found or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability.
        """
        self.resultSignal.emit("Starting VPC scan...")
        client = boto3.client('ec2', region_name='us-east-2')
        vpcs = client.describe_vpcs()['Vpcs']

        vulnerabilities = {
            'default_vpc': {'risk_level': 'Low', 'recommendation': 'Consider using a non-default VPC'},
        }
        vulnerable_vpcs = []

        for vpc in vpcs:
            self.resultSignal.emit(f"Checking VPC {vpc['VpcId']}")
            if vpc['IsDefault']:
                self.resultSignal.emit(f"-> Vulnerability: VPC {vpc['VpcId']} is the default VPC.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['default_vpc']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['default_vpc']['recommendation']}")
                vulnerable_vpcs.append(vpc['VpcId'])

        if vulnerable_vpcs:
            self.resultSignal.emit(f"Vulnerable VPCs found: {', '.join(vulnerable_vpcs)}")
        else:
            self.resultSignal.emit("No vulnerable VPCs found.")
    def scan_secrets_manager(self):
        """
        Scans AWS Secrets Manager for vulnerabilities, specifically focusing on secrets stored in plaintext.

        This function checks each secret managed by AWS Secrets Manager to determine if any contain sensitive
        information, like passwords, in plaintext. Storing secrets in plaintext is considered a high-risk vulnerability,
        as it can lead to unauthorized access and potential data breaches.

        The scan involves listing all secrets and then retrieving the actual secret value for each one.
        Secrets containing a plaintext password are reported as vulnerabilities, along with a risk level and a recommendation
        to avoid storing secrets in plaintext.

        Note:
        - The function uses boto3 to interact with the AWS Secrets Manager service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the Secrets Manager scan, reports the status of each secret,
          and provides a summary of all vulnerable secrets found or a message if none are found.
```

```
        - resultSignal with detailed messages for each identified vulnerability.
        """
        self.resultSignal.emit("Starting Secrets Manager scan...")
        client = boto3.client('secretsmanager', region_name='us-east-2')
        secrets = client.list_secrets()['SecretList']

        vulnerabilities = {
            'plaintext_secret': {'risk_level': 'High', 'recommendation': 'Do not store secrets in plaintext'},
        }
        vulnerable_secrets = []

        for secret in secrets:
            self.resultSignal.emit(f"Checking Secret {secret['Name']}")
            secret_value = client.get_secret_value(SecretId=secret['ARN'])

            if 'password' in secret_value['SecretString']:
                self.resultSignal.emit(f"-> Vulnerability: Secret {secret['Name']} contains a plaintext password.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['plaintext_secret']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['plaintext_secret']['recommendation']}")
                vulnerable_secrets.append(secret['Name'])

        if vulnerable_secrets:
            self.resultSignal.emit("Vulnerable Secrets found:", vulnerable_secrets)
        else:
            self.resultSignal.emit("No vulnerable Secrets found.")
    def scan_kms(self
        """
        Scans AWS Key Management Service (KMS) for keys that lack key rotation.

        This function iterates over all KMS keys and checks if key rotation is enabled for each key.
        Key rotation is an essential security practice for managing encryption keys. Keys without
        enabled key rotation are flagged as a vulnerability due to the increased risk of compromise
        over time.

        The scan involves listing all KMS keys and retrieving their metadata to check the status of
        key rotation. Keys without enabled key rotation are reported, along with a risk level and a
        recommendation to enable key rotation.

        Note:
        - The function uses boto3 to interact with the AWS KMS service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the
          'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the KMS scan and reports the status of each key.
        - resultSignal with messages for each key identified as lacking key rotation, including the
          risk level and recommendation.
        """
        self.resultSignal.emit("Starting KMS scan...")
        client = boto3.client('kms', region_name='us-east-2')
        keys = client.list_keys()['Keys']

        vulnerabilities = {
            'key_rotation': {'risk_level': 'Medium', 'recommendation': 'Enable key rotation'},
        }
        vulnerable_keys = []

        for key in keys:
            self.resultSignal.emit(f"Checking Key {key['KeyId']}")
            key_metadata = client.describe_key(KeyId=key['KeyId'])['KeyMetadata']
            if not key_metadata.get('KeyRotationEnabled', False):
                self.resultSignal.emit(f"-> Vulnerability: Key {key['KeyId']} does not have key rotation enabled.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['key_rotation']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['key_rotation']['recommendation']}")
                vulnerable_keys.append(key['KeyId'])

        if vulnerable_keys:
            self.resultSignal.emit("Vulnerable Keys found:", vulnerable_keys)
        else:
            self.resultSignal.emit("No vulnerable Keys found.")
    def scan_athena(self):
        """
        Scans AWS Athena workgroups for security vulnerabilities, specifically focusing on public result configurations.

        This function checks each Athena workgroup in the specified AWS region to determine if the workgroup configuration allows
        public access to query results. Public accessibility of query results is flagged as a vulnerability due to the potential
        exposure of sensitive data.

        The scan involves listing all Athena workgroups, retrieving their configurations, and checking for any configurations
        that allow public access to query results. Workgroups with such configurations are reported as vulnerabilities, along
        with a risk level and a recommendation to ensure result configurations are not public.

        Note:
        - The function uses boto3 to interact with the AWS Athena service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the Athena scan, reports the status of each workgroup, and provides a summary of all vulnerable workgroups found or a message if none are found
        - resultSignal with detailed messages for each identified vulnerability.
        """
        self.resultSignal.emit("Starting Athena scan...")
        client = boto3.client('athena', region_name='us-east-2')

        vulnerabilities = {
            'public_result_config': {'risk_level': 'Medium', 'recommendation': 'Ensure result configurations are not public'},
        }

        vulnerable_workgroups = []

        # List all workgroups
        workgroups = client.list_work_groups()['WorkGroups']

        for workgroup in workgroups:
            self.resultSignal.emit(f"Checking Workgroup {workgroup['Name']}")

            # Retrieve workgroup configuration (Assume 'IsResultPublic' is a field, which is not true in reality)
            workgroup_config = client.get_work_group(WorkGroup=workgroup['Name'])

            # Check if workgroup allows results to be public (Theoretical example)
            if workgroup_config.get('Configuration', {}).get('IsResultPublic', False):
                self.resultSignal.emit(f"-> Vulnerability: Workgroup {workgroup['Name']} allows public query results.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['public_result_config']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['public_result_config']['recommendation']}")
                vulnerable_workgroups.append(workgroup['Name'])

        if vulnerable_workgroups:
            self.resultSignal.emit("Vulnerable Athena Workgroups found:", vulnerable_workgroups)
        else:
            self.resultSignal.emit("No vulnerable Athena Workgroups found.")

    def scan_step_functions(self):
        """
        Scans AWS Step Functions for security vulnerabilities, specifically focusing on unrestricted execution permissions.

        This function checks each AWS Step Function state machine to determine if its associated IAM role has overly permissive
        policies. Unrestricted execution permissions in state machines are considered high-risk vulnerabilities as they can
        potentially allow unauthorized actions.

        The scan involves listing all state machines, describing each to obtain its associated IAM role, and then retrieving
        and evaluating the policies attached to these roles. State machines with roles that have overly permissive policies
        are reported as vulnerabilities, along with a risk level and a recommendation to restrict execution permissions.
```

```
        Note:
        - The function uses boto3 to interact with AWS Step Functions and IAM services.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the\
        'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the Step Functions scan, reports the status of each state machine,\
        and provides a summary of all vulnerable functions found or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability.
        """
        self.resultSignal.emit("Starting Step Functions scan...")
        client = boto3.client('stepfunctions', region_name='us-east-2')
        iam_client = boto3.client('iam')

        vulnerabilities = {
            'unrestricted_execution': {'risk_level': 'High', 'recommendation': 'Restrict Step Function execution permissions'},
        }
        vulnerable_functions = []

        # List all State Machines
        state_machines = client.list_state_machines()['stateMachines']

        for sm in state_machines:
            self.resultSignal.emit(f"Checking State Machine {sm['name']}")

            # Describe the State Machine to get its role ARN
            sm_description = client.describe_state_machine(stateMachineArn=sm['stateMachineArn'])
            role_arn = sm_description['roleArn']

            # Assume the role ARN follows the pattern 'arn:aws:iam::account-ID-without-hyphens:role/role-name'
            role_name = role_arn.split('/')[-1]

            # Retrieve IAM role policies (only inline policies in this example)
            inline_policies = iam_client.list_role_policies(RoleName=role_name)['PolicyNames']

            for policy_name in inline_policies:
                policy_document = iam_client.get_role_policy(RoleName=role_name, PolicyName=policy_name)['PolicyDocument']
                policy_json = json.dumps(policy_document)

                # Check for overly permissive policies (e.g., "Resource": "*")
                if '"Resource": "*"' in policy_json:
                    self.resultSignal.emit(f"-> Vulnerability: State Machine {sm['name']} \
                    has unrestricted execution permissions.")
                    self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['unrestricted_execution']['risk_level']}")
                    self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['unrestricted_execution']['recommendation']}")
                    vulnerable_functions.append(sm['name'])
                    break  # Break out of the inline policies loop, as we found a vulnerability

        if vulnerable_functions:
            self.resultSignal.emit("Vulnerable Step Functions found:", vulnerable_functions)
        else:
            self.resultSignal.emit("No vulnerable Step Functions found.")
    def scan_redshift(self):
        """
        Scans AWS Redshift clusters for security vulnerabilities, specifically focusing on public accessibility.

        This function iterates over all Redshift clusters in the specified AWS region and checks if they are publicly \
        accessible.
        Public accessibility of Redshift clusters is considered a high-risk vulnerability due to the potential exposure of
        sensitive data.

        The scan involves listing all Redshift clusters and checking their 'PubliclyAccessible' attribute. Clusters that are
        publicly accessible are reported as vulnerabilities, along with a risk level and a recommendation to ensure the \
        clusters are not publicly accessible.

        Note:
        - The function uses boto3 to interact with the AWS Redshift service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' \
        dictionary.

        Emits:
        - resultSignal: Signals the start of the Redshift scan, reports the status of each cluster, and provides a summary of
          all vulnerable clusters found or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability.
        """
        self.resultSignal.emit("Starting Redshift scan...")
        client = boto3.client('redshift', region_name='us-east-2')

        clusters = client.describe_clusters()['Clusters']
        vulnerabilities = {
            'public_cluster': {'risk_level': 'High', 'recommendation': 'Ensure Redshift clusters are not publicly accessible'},
        }
        vulnerable_clusters = []

        for cluster in clusters:
            self.resultSignal.emit(f"Checking Cluster {cluster['ClusterIdentifier']}")
            if cluster.get('PubliclyAccessible'):
                self.resultSignal.emit(f"-> Vulnerability: Cluster {cluster['ClusterIdentifier']} is publicly accessible.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['public_cluster']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['public_cluster']['recommendation']}")
                vulnerable_clusters.append(cluster['ClusterIdentifier'])

        if vulnerable_clusters:
            self.resultSignal.emit("Vulnerable Redshift Clusters found:", vulnerable_clusters)
        else:
            self.resultSignal.emit("No vulnerable Redshift Clusters found.")

    def scan_sagemaker(self
        """
        Scans AWS SageMaker notebook instances for security vulnerabilities, focusing on direct internet access.

        This function checks each SageMaker notebook instance to determine if it has direct internet access enabled.
        Enabling direct internet access can be a security risk, potentially exposing the notebook instance to
        unauthorized access.

        The scan involves listing all SageMaker notebook instances and checking their 'DirectInternetAccess' attribute.
        Instances with direct internet access enabled are reported as vulnerabilities, along with a risk level and a
        recommendation to disable direct internet access.

        Note:
        - The function uses boto3 to interact with the AWS SageMaker service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' \
        dictionary.

        Emits:
        - resultSignal: Signals the start of the SageMaker scan, reports the status of each notebook instance, and provides
          a summary of all vulnerable notebooks found or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability.
        """
        self.resultSignal.emit("Starting SageMaker scan...")
        client = boto3.client('sagemaker', region_name='us-east-2')

        vulnerabilities = {
            'public_notebook': {'risk_level': 'High', 'recommendation': 'Disable direct internet access to the notebook.'},
        }
        vulnerable_notebooks = []

        # List all Notebook instances
        notebook_instances = client.list_notebook_instances()['NotebookInstances']

        for notebook in notebook_instances:
            if notebook.get('DirectInternetAccess') == 'Enabled':
```

```
            vulnerable_notebooks.append(notebook['NotebookInstanceName'])
            self.resultSignal.emit(f"-> Vulnerability: Notebook {notebook['NotebookInstanceName']} \
            has direct internet access.")
            self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['public_notebook']['risk_level']}")
            self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['public_notebook']['recommendation']}")

    if vulnerable_notebooks:
        self.resultSignal.emit("Vulnerable Notebooks found:", vulnerable_notebooks)
    else:
        self.resultSignal.emit("No vulnerable Notebooks found.")
def scan_api_gateway(self):
    """
    Scans AWS API Gateway for security vulnerabilities, specifically focusing on unrestricted Cross-Origin Resource Sharing (CORS) configurations.

    This function checks each API in the AWS API Gateway service to determine if any of them have open or unrestricted CORS settings.
    Unrestricted CORS configurations can pose security risks by allowing potentially unsafe cross-origin requests.

    The scan involves listing all REST APIs, retrieving their resources and methods, and then checking the CORS configurations in method responses.
    APIs that allow unrestricted access in their CORS configurations are reported as vulnerabilities, along with a risk level and a recommendation
    to restrict CORS headers.

    Note:
    - The function uses boto3 to interact with the AWS API Gateway service.
    - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' dictionary.

    Emits:
    - resultSignal: Signals the start of the API Gateway scan, reports the status of each API, and provides a summary of all vulnerable APIs found or a message if none are found.
    - resultSignal with detailed messages for each identified vulnerability.
    """
    self.resultSignal.emit("Starting API Gateway scan...")
    client = boto3.client('apigateway', region_name='us-east-2')

    vulnerabilities = {
        'open_cors': {'risk_level': 'Medium', 'recommendation': 'Restrict CORS headers'},
    }
    vulnerable_apis = []

    # List all Rest APIs
    rest_apis = client.get_rest_apis()['items']

    for api in rest_apis:
        resources = client.get_resources(restApiId=api['id'])['items']
        for resource in resources:
            methods = resource.get('resourceMethods', {})
            for method, config in methods.items():
                if 'methodResponses' in config:
                    for response in config['methodResponses'].values():
                        if '*' in response.get('responseParameters', {}).\
                        get('method.response.header.Access-Control-Allow-Origin', ''):
                            self.resultSignal.emit(f"-> Vulnerability: API {api['name']} allows unrestricted CORS.")
                            self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['open_cors']['risk_level']}")
                            self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['open_cors']['recommendation']}")
                            vulnerable_apis.append(api['id'])
                            break

    if vulnerable_apis:
        self.resultSignal.emit("Vulnerable APIs found:", vulnerable_apis)
    else:
        self.resultSignal.emit("No vulnerable APIs found.")

def scan_cognito(self):
    """
    Scans AWS Cognito user pools for vulnerabilities related to weak password policies.

    This function examines each Cognito user pool to check the strength of its password policy.
    Weak password policies, characterized by insufficient minimum length and lack of complexity requirements,
    are flagged as high-risk vulnerabilities. This is due to the increased likelihood of password compromise.

    The scan involves listing all user pools, retrieving their password policies, and assessing the policies' strength
    based on predefined criteria, such as minimum length. User pools not meeting these criteria are reported as
    vulnerabilities, along with a risk level and a recommendation to strengthen the password policy.

    Note:
    - The function uses boto3 to interact with the AWS Cognito Identity Provider service.
    - Vulnerabilities, their risk levels, and recommendations are predefined in the 'vulnerabilities' dictionary.

    Emits:
    - resultSignal: Signals the start of the Cognito scan, reports the status of each user pool, and provides a
      summary of all vulnerable user pools found or a message if none are found.
    - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
"""
    self.resultSignal.emit("Starting Cognito scan...")
    client = boto3.client('cognito-idp', region_name='us-east-2')

    vulnerabilities = {
        'weak_password_policy': {'risk_level': 'High', 'recommendation': 'Strengthen password policy'},
    }
    vulnerable_pools = []

    # List all user pools
    user_pools = client.list_user_pools(MaxResults=10)['UserPools']

    for pool in user_pools:
        policy = client.describe_user_pool(UserPoolId=pool['Id'])['UserPool'].get('Policies', {}).get('PasswordPolicy', {})

        if not policy.get('MinimumLength', 0) >= 12:
            self.resultSignal.emit(f"-> Vulnerability: User pool {pool['Name']} has a weak password policy.")
            self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['weak_password_policy']['risk_level']}")
            self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['weak_password_policy']['recommendation']}")
            vulnerable_pools.append(pool['Id'])

    if vulnerable_pools:
        self.resultSignal.emit("Vulnerable User Pools found:", vulnerable_pools)
    else:
        self.resultSignal.emit("No vulnerable User Pools found.")

def scan_route_53(self):
    """
    Scans AWS Route 53 for domains lacking transfer lock, identifying potential vulnerabilities.

    This function assesses each domain within AWS Route 53's hosted zones to check if domain transfer locks are enabled.
    The absence of a transfer lock is flagged as a high-risk vulnerability, as it could lead to unauthorized domain transfers.

    The scan involves listing all hosted zones, retrieving detailed information for each domain, and then verifying the
    presence of a transfer lock. Domains without transfer locks are reported as vulnerabilities, along with a risk level and
    a recommendation to enable the transfer lock for enhanced security.

    Note:
    - The function uses boto3 to interact with the AWS Route 53 service.
    - Vulnerabilities, their risk levels, and recommendations are predefined in the 'vulnerabilities' dictionary.

    Emits:
    - resultSignal: Signals the start of the Route 53 scan, reports the status of each domain, and provides a summary of all
      vulnerable domains found, or a message if none are found.
    - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
    """
    self.resultSignal.emit("Starting Route 53 scan...")
    client = boto3.client('route53', region_name='us-east-2')

    vulnerabilities = {
        'domain_transfer_lock_disabled': {'risk_level': 'High', 'recommendation': 'Enable domain transfer lock'},
    }
```

```
        vulnerable_domains = []

        # List all hosted zones
        hosted_zones = client.list_hosted_zones()['HostedZones']

        for zone in hosted_zones:
            domain_details = client.get_domain_detail(DomainName=zone['Name'])
            if not domain_details.get('TransferLock'):
                self.resultSignal.emit(f"-> Vulnerability: Domain {zone['Name']} does not have transfer lock enabled.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['domain_transfer_lock_disabled']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: \
                {vulnerabilities['domain_transfer_lock_disabled']['recommendation']}")
                vulnerable_domains.append(zone['Id'])

        if vulnerable_domains:
            self.resultSignal.emit("Vulnerable Domains found:", vulnerable_domains)
        else:
            self.resultSignal.emit("No vulnerable Domains found.")

    def scan_waf(self):
        """
        Scans AWS Web Application Firewall (WAF) for vulnerabilities related to default actions set to 'ALLOW'.

        This function examines each Web Access Control List (WebACL) in AWS WAF to identify if the default action is set to 'ALLOW'.
        Such a configuration is considered a high-risk vulnerability as it can permit potentially harmful traffic by default.

        The scan involves listing all WebACLs, retrieving their details, and checking the 'DefaultAction' setting. WebACLs with the
        default action set to 'ALLOW' are reported as vulnerabilities, along with a risk level and a recommendation to change the
        default action to 'BLOCK'.

        Note:
        - The function uses boto3 to interact with the AWS WAF service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the `vulnerabilities` dictionary.

        Emits:
        - resultSignal: Signals the start of the WAF scan, reports the status of each WebACL, and provides a summary of all
          vulnerable WebACLs found or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting WAF scan...")
        client = boto3.client('waf', region_name='us-east-2')

        vulnerabilities = {
            'default_action_allow': {'risk_level': 'High', 'recommendation': 'Change default action to block'},
        }
        vulnerable_wafs = []

        # List WebACLs
        web_acls = client.list_web_acls()['WebACLs']

        for acl in web_acls:
            details = client.get_web_acl(WebACLId=acl['WebACLId'])['WebACL']
            if details['DefaultAction']['Type'] == 'ALLOW':
                self.resultSignal.emit(f"-> Vulnerability: WebACL {acl['Name']} has default action set to ALLOW.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['default_action_allow']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['default_action_allow']['recommendation']}")
                vulnerable_wafs.append(acl['WebACLId'])

        if vulnerable_wafs:
            self.resultSignal.emit("Vulnerable WebACLs found:", vulnerable_wafs)
        else:
            self.resultSignal.emit("No vulnerable WebACLs found.")

    def scan_inspector(self):
        """
        Scans AWS Inspector for vulnerabilities, focusing on outdated assessment templates.

        This function assesses each AWS Inspector assessment template to determine if they are outdated, based on a predefined
        date criterion. Using old assessment templates can be a medium-risk vulnerability, as it may lead to less effective
        security assessments due to outdated criteria or configurations.

        The scan involves listing all assessment templates, retrieving their details, and checking their creation date against
        a predefined threshold. Templates created before this threshold are reported as vulnerabilities, along with a risk level
        and a recommendation to update the assessment template.

        Note:
        - The function uses boto3 to interact with the AWS Inspector service.
        - The 'some_old_date' threshold should be defined to determine what constitutes an 'old' template.
        - Vulnerabilities, their risk levels, and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the Inspector scan, reports the status of each assessment template, and provides a
          summary of all vulnerable templates found, or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting Inspector scan...")
        client = boto3.client('inspector', region_name='us-east-2')

        vulnerabilities = {
            'old_assessment_template': {'risk_level': 'Medium', 'recommendation': 'Consider updating the assessment template'},
        }
        vulnerable_templates = []

        # List assessment templates
        templates = client.list_assessment_templates()['assessmentTemplateArns']

        for template_arn in templates:
            details = client.describe_assessment_templates(assessmentTemplateArns=[template_arn])['assessmentTemplates'][0]
            if details['createdAt'] < some_old_date:
                self.resultSignal.emit(f"-> Vulnerability: Assessment Template {details['name']} is old.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['old_assessment_template']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['old_assessment_template']['recommendation']}")
                vulnerable_templates.append(template_arn)

        if vulnerable_templates:
            self.resultSignal.emit("Vulnerable Assessment Templates found:", vulnerable_templates)
        else:
            self.resultSignal.emit("No vulnerable Assessment Templates found.")

    def scan_glue(self):
        """
        Scans AWS Glue for vulnerabilities, specifically focusing on jobs that are publicly accessible.

        This function checks each AWS Glue job to determine if its role is set to 'Public'. Publicly accessible Glue jobs
        are considered a high-risk vulnerability due to potential unauthorized access or data exposure.

        The scan involves listing all Glue jobs and checking their role settings. Jobs identified as 'Public' are reported as
        vulnerabilities, with corresponding risk levels and recommendations for remediation, such as making the job private.

        Note:
        - The function uses boto3 to interact with the AWS Glue service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the Glue scan, reports the status of each job, and provides a summary of all
          vulnerable jobs found, or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting Glue scan...")
        client = boto3.client('glue', region_name='us-east-2')

        vulnerabilities = {
```

```
            'public_job': {'risk_level': 'High', 'recommendation': 'Make the Glue job private'},
        }
        vulnerable_jobs = []

        # List Glue jobs
        jobs = client.get_jobs()['Jobs']

        for job in jobs:
            if job.get('Role') == 'Public':
                self.resultSignal.emit(f"-> Vulnerability: Glue job {job['Name']} is public.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['public_job']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['public_job']['recommendation']}")
                vulnerable_jobs.append(job['Name'])

        if vulnerable_jobs:
            self.resultSignal.emit("Vulnerable Glue Jobs found:", vulnerable_jobs)
        else:
            self.resultSignal.emit("No vulnerable Glue Jobs found.")


    def scan_ec2_auto_scaling(self):
        """
        Scans AWS EC2 Auto Scaling groups for vulnerabilities, focusing on insecure launch configurations.

        This function evaluates each Auto Scaling group to identify if it is using a launch configuration deemed 'insecure'.
        Insecure launch configurations in Auto Scaling groups are considered a medium-risk vulnerability, as they can lead to
        the deployment of instances with potentially vulnerable or non-compliant settings.

        The scan process involves listing all Auto Scaling groups and examining their launch configuration names. Groups with
        a launch configuration name identified as 'insecure-config' are reported as vulnerabilities, along with a risk \
        level and
        a recommendation for using more secure configurations.

        Note:
        - The function uses boto3 to interact with the AWS Auto Scaling service.
        - The criteria for what constitutes an 'insecure' configuration should be clearly defined and updated as needed.
        - Vulnerabilities, their risk levels, and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the EC2 Auto Scaling scan, reports the status of each group, and provides \
        a summary
          of all vulnerable groups found, or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting EC2 Auto Scaling scan...")
        client = boto3.client('autoscaling', region_name='us-east-2')

        vulnerabilities = {
            'insecure_launch_config': {'risk_level': 'Medium', 'recommendation': 'Use secure launch configurations'},
        }
        vulnerable_groups = []

        # List Auto Scaling groups
        groups = client.describe_auto_scaling_groups()['AutoScalingGroups']

        for group in groups:
            if group.get('LaunchConfigurationName') == 'insecure-config':
                self.resultSignal.emit(f"-> Vulnerability: Auto Scaling Group {group['AutoScalingGroupName']}\
                uses insecure launch configuration.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['insecure_launch_config']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['insecure_launch_config']['recommendation']}")
                vulnerable_groups.append(group['AutoScalingGroupName'])

        if vulnerable_groups:
            self.resultSignal.emit("Vulnerable Auto Scaling Groups found:", vulnerable_groups)
        else:
            self.resultSignal.emit("No vulnerable Auto Scaling Groups found.")

    def scan_ecs(self):
        """
        Scans AWS Elastic Container Service (ECS) for tasks running in a potentially vulnerable network mode.

        This function inspects each ECS task definition to check if it is running in 'host' network mode. Tasks running in
        'host' network mode are considered a high-risk vulnerability as they share the network namespace with the host,
        potentially leading to security risks like unintended network access or conflicts.

        The scan process involves listing all ECS task definitions and examining their network mode. Tasks running in the
        'host' network mode are identified as vulnerabilities and reported with a risk level and a recommendation to use a
        more secure network configuration.

        Note:
        - The function uses boto3 to interact with the AWS ECS service.
        - Vulnerabilities, their risk levels, and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the ECS scan, reports the status of each task definition, and provides a summary
          of all vulnerable tasks found, or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
    """
        self.resultSignal.emit("Starting ECS scan...")
        client = boto3.client('ecs', region_name='us-east-2')

        vulnerabilities = {
            'public_task': {'risk_level': 'High', 'recommendation': 'Make the task definition private'},
        }
        vulnerable_tasks = []

        # List task definitions
        task_definitions = client.list_task_definitions()['taskDefinitionArns']

        for task_arn in task_definitions:
            task = client.describe_task_definition(taskDefinition=task_arn)['taskDefinition']
            if task.get('networkMode') == 'host':
                self.resultSignal.emit(f"-> Vulnerability: ECS Task {task['taskDefinitionArn']} \
                is running in host network mode.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['public_task']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['public_task']['recommendation']}")
                vulnerable_tasks.append(task['taskDefinitionArn'])

        if vulnerable_tasks:
            self.resultSignal.emit("Vulnerable ECS Tasks found:", vulnerable_tasks)
        else:
            self.resultSignal.emit("No vulnerable ECS Tasks found.")

    def scan_ecr(self):
        """
        Scans AWS Elastic Container Registry (ECR) for public repositories, identifying potential security vulnerabilities.

        This function reviews each ECR repository to determine if its policy settings allow public access, which is considered
        a high-risk vulnerability due to the potential exposure of sensitive data or container images.

        The scan involves using pagination to handle accounts with a large number of repositories, examining each repository's
        policy settings. Repositories with policies granting 'Allow' access to all principals ('*') are identified as public \
        and
        therefore vulnerable. These are reported with a risk level and a recommendation to restrict access by making the
        repository private.

        Note:
        - The function uses boto3 to interact with the AWS ECR service.
        - Vulnerabilities and their respective risk levels and recommendations are predefined in the 'vulnerabilities' \
        dictionary.
        - Handles potential ClientErrors during repository policy retrieval.
```

```
        Emits:
        - resultSignal: Signals the start of the ECR scan, reports the status of each repository, and provides a summary
          of all vulnerable repositories found, or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting ECR scan...")
        client = boto3.client('ecr', region_name='us-east-2')

        vulnerabilities = {
            'public_repo': {'risk_level': 'High', 'recommendation': 'Make the ECR repository private'},
        }
        vulnerable_repos = []

        try:
            # Handle pagination for accounts with more than 1000 repositories
            paginator = client.get_paginator('describe_repositories')
            for page in paginator.paginate():
                for repo in page['repositories']:
                    try:
                        policy_response = client.get_repository_policy(repositoryName=repo['repositoryName'])
                        policy_text = policy_response['policyText']
                        policy = json.loads(policy_text)

                        # Check if the repository is public
                        for statement in policy.get('Statement', []):
                            if statement.get('Effect') == 'Allow' and statement.get('Principal') == '*':
                                self.resultSignal.emit(f"-> Vulnerability: ECR repository {repo['repositoryName']} is public.")
                                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['public_repo']['risk_level']}")
                                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['public_repo']['recommendation']}")
                                vulnerable_repos.append(repo['repositoryName'])

                    except ClientError as e:
                        self.resultSignal.emit(f"Could not get policy for repository {repo['repositoryName']}. Error: {e}")

        except ClientError as e:
            self.resultSignal.emit(f"Could not describe repositories. Error: {e}")

        if vulnerable_repos:
            self.resultSignal.emit("Vulnerable ECR Repositories found:", vulnerable_repos)
        else:
            self.resultSignal.emit("No vulnerable ECR Repositories found.")

    def scan_swf(self):
        """
        Performs a security scan on AWS Simple Workflow Service (SWF) domains to identify those allowing open registration.

        This function checks each registered SWF domain to see if it allows open registration. Domains with open registration
        are considered medium-risk vulnerabilities, as they may permit unauthorized entities to register workflows or
        activities, potentially leading to unauthorized access or misuse of the SWF service.

        The scan involves listing all registered SWF domains and checking their registration status. Domains that are flagged
        as allowing open registration are reported as vulnerabilities, with associated risk levels and recommendations to
        restrict domain registration.

        Note:
        - The function uses boto3 to interact with the AWS SWF service.
        - Vulnerabilities, their risk levels, and recommendations are predefined in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the initiation of the SWF scan, reports on each domain's status, and provides a summary of all
          vulnerable domains found or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting SWF scan...")
        client = boto3.client('swf', region_name='us-east-2')

        vulnerabilities = {
            'open_registration': {'risk_level': 'Medium', 'recommendation': 'Restrict domain registration'},
        }
        vulnerable_domains = []

        # List SWF domains
        domains = client.list_domains(registrationStatus='REGISTERED')['domainInfos']

        for domain in domains:
            if domain.get('status') == 'REGISTERED':
                self.resultSignal.emit(f"-> Vulnerability: SWF domain {domain['name']} allows open registration.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['open_registration']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['open_registration']['recommendation']}")
                vulnerable_domains.append(domain['name'])

        if vulnerable_domains:
            self.resultSignal.emit("Vulnerable SWF Domains found:", vulnerable_domains)
        else:
            self.resultSignal.emit("No vulnerable SWF Domains found.")

    def scan_elastic_transcoder(self
        """
        Scans AWS Elastic Transcoder pipelines for vulnerabilities related to missing notifications.

        This function evaluates each Elastic Transcoder pipeline to determine if it is configured with notifications.
        The absence of notifications in a pipeline is considered a low-risk vulnerability. Notifications are crucial
        for monitoring pipeline activities and receiving alerts about processing issues, ensuring timely response and
        intervention.

        The scan process involves listing all Elastic Transcoder pipelines and checking each for its notification setup.
        Pipelines without any notification configuration are reported as vulnerabilities, with a low risk level and a
        recommendation to set up appropriate notifications.

        Note:
        - The function uses boto3 to interact with the AWS Elastic Transcoder service.
        - Predefined vulnerabilities, risk levels, and recommendations are provided in the 'vulnerabilities' dictionary.

        Emits:
        - resultSignal: Signals the start of the scan, reports the status of each pipeline, and provides a summary of all
          vulnerable pipelines found, or a message if none are found.
        - resultSignal with detailed messages for each identified vulnerability, including the risk level and recommendation.
        """
        self.resultSignal.emit("Starting Elastic Transcoder scan...")
        client = boto3.client('elastictranscoder', region_name='us-east-2')

        vulnerabilities = {
            'no_notifications': {'risk_level': 'Low', 'recommendation': 'Set up notifications for the pipeline'},
        }
        vulnerable_pipelines = []

        # List Elastic Transcoder pipelines
        pipelines = client.list_pipelines()['Pipelines']

        for pipeline in pipelines:
            if not pipeline.get('Notifications'):
                self.resultSignal.emit(f"-> Vulnerability: Elastic Transcoder pipeline {pipeline['Name']} \
                does not have notifications set.")
                self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['no_notifications']['risk_level']}")
                self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['no_notifications']['recommendation']}")
                vulnerable_pipelines.append(pipeline['Name'])

        if vulnerable_pipelines:
            self.resultSignal.emit("Vulnerable Elastic Transcoder Pipelines found:", vulnerable_pipelines)
        else:
            self.resultSignal.emit("No vulnerable Elastic Transcoder Pipelines found.")
```

```
def scan_elastic_beanstalk(self
    """
    Conducts a scan of AWS Elastic Beanstalk environments to identify those with public URLs, potentially exposing them\
    to security risks.

    This function evaluates each Elastic Beanstalk environment to check if it has a publicly accessible URL. Environments
    with public URLs are considered medium-risk vulnerabilities because they could be exposed to unauthorized access or
    attacks from the internet. The function focuses on identifying these environments and suggesting a remediation approach.

    The scanning process involves listing all Elastic Beanstalk environments and inspecting each for the presence of a
    public CNAME (Canonical Name). Environments with a CNAME are reported as having public URLs, accompanied by a risk
    level and a recommendation to disable public access.

    Note:
    - This function utilizes boto3 for interacting with the AWS Elastic Beanstalk service.
    - The function defines vulnerabilities and their respective risk levels and recommendations in the 'vulnerabilities'\
    dictionary.

    Emits:
    - resultSignal: Signaling the start of the scan and reporting on the status of each environment. It provides a summary
      of all vulnerable environments or a message indicating none were found.
    - resultSignal: Emitting detailed messages for each identified vulnerability, including the risk level and recommended \
    actions.
    """
    self.resultSignal.emit("Starting Elastic Beanstalk scan...")
    client = boto3.client('elasticbeanstalk', region_name='us-east-2')

    vulnerabilities = {
        'public_url': {'risk_level': 'Medium', 'recommendation': 'Disable public access'},
    }
    vulnerable_environments = []

    # List all environments
    environments = client.describe_environments()['Environments']

    for env in environments:
        if 'CNAME' in env:
            self.resultSignal.emit(f"-> Vulnerability: Environment {env['EnvironmentName']} has a public URL ({env['CNAME']}).")
            self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['public_url']['risk_level']}")
            self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['public_url']['recommendation']}")
            vulnerable_environments.append(env['EnvironmentName'])

    if vulnerable_environments:
        self.resultSignal.emit("Vulnerable Elastic Beanstalk Environments found:", vulnerable_environments)
    else:
        self.resultSignal.emit("No vulnerable Elastic Beanstalk Environments found.")

def scan_cloudtrail(self):
    """
    Executes a security scan on AWS CloudTrail trails to identify trails where logging is disabled.

    This function checks each CloudTrail trail to ascertain if logging is enabled. Trails with logging disabled
    are considered high-risk vulnerabilities, as logging is critical for monitoring and auditing AWS account activity.
    The absence of logging can lead to gaps in understanding user activities and potential security incidents.

    The scan involves listing all CloudTrail trails and inspecting their logging status. Trails with logging disabled
    are reported as vulnerabilities, each accompanied by a high risk level and a recommendation to enable logging.

    Note:
    - Utilizes boto3 for interaction with the AWS CloudTrail service.
    - Vulnerabilities are predefined in the 'vulnerabilities' dictionary, with associated risk levels and recommendations.

    Emits:
    - resultSignal: Announces the start of the CloudTrail scan, reports the status of each trail, and provides a summary
      of vulnerable trails or a message if no vulnerabilities are found.
    - resultSignal with detailed messages for each vulnerability, including risk assessment and advised remedial actions.
    """
    self.resultSignal.emit("Starting CloudTrail scan...")
    client = boto3.client('cloudtrail', region_name='us-east-2')

    vulnerabilities = {
        'logging_disabled': {'risk_level': 'High', 'recommendation': 'Enable logging'},
    }
    vulnerable_trails = []

    # List trails and check their status
    trails = client.describe_trails()['trailList']

    for trail in trails:
        status = client.get_trail_status(Name=trail['Name'])
        if not status['IsLogging']:
            self.resultSignal.emit(f"-> Vulnerability: Logging is disabled for trail {trail['Name']}.")
            self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['logging_disabled']['risk_level']}")
            self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['logging_disabled']['recommendation']}")
            vulnerable_trails.append(trail['Name'])

    if vulnerable_trails:
        self.resultSignal.emit("Vulnerable CloudTrails found:", vulnerable_trails)
    else:
        self.resultSignal.emit("No vulnerable CloudTrails found.")
def scan_cloudformation(self):
    """
    Performs a scan on AWS CloudFormation stacks to identify those in a failed state.

    This function evaluates CloudFormation stacks to detect any that are in a failed state, such as 'ROLLBACK_FAILED',
    'ROLLBACK_COMPLETE', or 'DELETE_FAILED'. Stacks in these states are considered vulnerabilities, albeit with a
    low risk level, as they might indicate unsuccessful operations or misconfigurations.

    The scan involves listing all CloudFormation stack summaries and checking their current status. Each stack in a
    failed state is reported, along with the specific failure status, a low-risk designation, and a recommendation for
    further investigation to understand and rectify the underlying issues.

    Note:
    - Uses boto3 to interface with the AWS CloudFormation service.
    - The 'vulnerabilities' dictionary predefines the risk levels and recommendations for identified vulnerabilities.

    Emits:
    - resultSignal: Signals the initiation of the CloudFormation scan, details the status of each stack, and provides a
      summary of vulnerable stacks or a notification if none are found.
    - resultSignal: Emits messages for each identified vulnerability, including risk level and recommended actions.
    """
    self.resultSignal.emit("Starting CloudFormation scan...")
    client = boto3.client('cloudformation', region_name='us-east-2')

    vulnerabilities = {
        'failed_state': {'risk_level': 'Low', 'recommendation': 'Investigate the failure'},
    }
    vulnerable_stacks = []

    # List stacks
    stacks = client.list_stacks()['StackSummaries']

    for stack in stacks:
        if stack['StackStatus'] in ['ROLLBACK_FAILED', 'ROLLBACK_COMPLETE', 'DELETE_FAILED']:
            self.resultSignal.emit(f"-> Vulnerability: Stack {stack['StackName']} \
            is in a failed state ({stack['StackStatus']}).")
            self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['failed_state']['risk_level']}")
            self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['failed_state']['recommendation']}")
            vulnerable_stacks.append(stack['StackName'])
```

```
        if vulnerable_stacks:
            self.resultSignal.emit("Vulnerable CloudFormation Stacks found:", vulnerable_stacks)
        else:
            self.resultSignal.emit("No vulnerable CloudFormation Stacks found.")

    def scan_codedeploy(self):
        """
        Scans AWS CodeDeploy deployment groups to identify those without auto-rollback enabled.

        This function iterates through all AWS CodeDeploy applications and their respective deployment groups. It checks
        whether auto-rollback is enabled for each deployment group. Auto-rollback is a crucial feature that automatically
        reverts the deployment process if certain conditions are not met or if failures are detected. Deployment groups
        without this feature enabled are considered vulnerable, with a medium risk level.

        The scan reports each vulnerable deployment group, specifying the application it belongs to and noting the absence
        of auto-rollback configuration. The method concludes by providing a summary of all identified vulnerabilities or
        confirms if no vulnerabilities were found.

        Note:
        - Utilizes boto3 for interactions with the AWS CodeDeploy service.
        - The 'vulnerabilities' dictionary defines the risk levels and recommendations for the identified issues.

        Emits:
        - resultSignal: Signals the start of the CodeDeploy scan, reports the status of each deployment group, and provides
          a summary of vulnerable groups or a notification if none are found.
        - resultSignal: Emits detailed messages for each identified vulnerability, including the affected deployment group,
          associated risk level, and recommended actions.
        """
        self.resultSignal.emit("Starting CodeDeploy scan...")
        client = boto3.client('codedeploy', region_name='us-east-2')

        vulnerabilities = {
            'no_auto_rollback': {'risk_level': 'Medium', 'recommendation': 'Enable auto-rollback for deployment groups'},
        }
        vulnerable_deploy_groups = []

        # List applications
        applications = client.list_applications()['applications']

        for app in applications:
            # List deployment groups for each application
            deploy_groups = client.list_deployment_groups(applicationName=app)['deploymentGroups']

            for group in deploy_groups:
                # Get deployment group details
                deploy_group_info = client.get_deployment_group(
                    applicationName=app,
                    deploymentGroupName=group
                )['deploymentGroupInfo']

                # Check if auto-rollback is enabled
                if 'autoRollbackConfiguration' in deploy_group_info:
                    auto_rollback_config = deploy_group_info['autoRollbackConfiguration']
                    if not auto_rollback_config.get('enabled', False):
                        self.resultSignal.emit(f"-> Vulnerability: Auto-rollback is not enabled for deployment group\
                        {group} in application {app}.")
                        self.resultSignal.emit(f"   - Risk Level: {vulnerabilities['no_auto_rollback']['risk_level']}")
                        self.resultSignal.emit(f"   - Recommendation: {vulnerabilities['no_auto_rollback']['recommendation']}")
                        vulnerable_deploy_groups.append({'application': app, 'deployment_group': group})

        if vulnerable_deploy_groups:
            self.resultSignal.emit("Vulnerable CodeDeploy Deployment Groups found:", vulnerable_deploy_groups)
        else:
            self.resultSignal.emit("No vulnerable CodeDeploy Deployment Groups found.")
class DualListWidget(QWidget):
    """
    A custom widget that provides a dual list interface for moving items between two lists.

    This widget contains two QListWidgets: one for available items and one for selected items.
    It also includes buttons to move items between these lists, either individually or in bulk.

    Attributes:
        availableList (QListWidget): A list widget to display available items.
        selectedList (QListWidget): A list widget to display selected items.
        btnAdd (QPushButton): A button to move selected items to the 'selectedList'.
        btnRemove (QPushButton): A button to move selected items back to the 'availableList'.
        btnAddAll (QPushButton): A button to move all items to the 'selectedList'.
        btnRemoveAll (QPushButton): A button to move all items back to the 'availableList'.
    """
    def __init__(self, parent=None):
        super().__init__(parent)

        layout = QHBoxLayout()

        # Available services list
        self.availableList = QListWidget(self)
        layout.addWidget(self.availableList)

        # Buttons for adding/removing services with annotations
        self.btnLayout = QVBoxLayout()

        # Button palette for clear visibility
        button_palette = QPalette()
        button_palette.setColor(QPalette.ButtonText, Qt.black)  # Setting the text color to black

        # Add annotation and button
        self.addLabel = QLabel("Add", self)
        self.btnAdd = QPushButton(">", self)
        self.btnAdd.setPalette(button_palette)
        self.btnAdd.clicked.connect(self.move_to_selected)
        self.btnLayout.addWidget(self.addLabel)
        self.btnLayout.addWidget(self.btnAdd)

        # Remove annotation and button
        self.removeLabel = QLabel("Remove", self)
        self.btnRemove = QPushButton("<", self)
        self.btnRemove.setPalette(button_palette)
        self.btnRemove.clicked.connect(self.move_to_available)
        self.btnLayout.addWidget(self.removeLabel)
        self.btnLayout.addWidget(self.btnRemove)

        # Add All annotation and button
        self.addAllLabel = QLabel("Add All", self)
        self.btnAddAll = QPushButton(">>", self)
        self.btnAddAll.setPalette(button_palette)
        self.btnAddAll.clicked.connect(self.move_all_to_selected)
        self.btnLayout.addWidget(self.addAllLabel)
        self.btnLayout.addWidget(self.btnAddAll)

        # Remove All annotation and button
        self.removeAllLabel = QLabel("Remove All", self)
        self.btnRemoveAll = QPushButton("<<", self)
        self.btnRemoveAll.setPalette(button_palette)
        self.btnRemoveAll.clicked.connect(self.move_all_to_available)
        self.btnLayout.addWidget(self.removeAllLabel)
        self.btnLayout.addWidget(self.btnRemoveAll)

        layout.addLayout(self.btnLayout)

        # Selected services list
        self.selectedList = QListWidget(self)
        layout.addWidget(self.selectedList)
```

```
        self.setLayout(layout)

    # Available services list

    def populate_available_services(self, items):
        self.availableList.addItems(items)

    def move_to_selected(self):
        """ Moves selected items from the available list to the selected list. """
        items = [self.availableList.takeItem(index.row()) for index in self.availableList.selectedIndexes()]
        for item in items:
            self.selectedList.addItem(item)

    def move_to_available(self):
        items = [self.selectedList.takeItem(index.row()) for index in self.selectedList.selectedIndexes()]
        for item in items:
            self.availableList.addItem(item)

    def move_all_to_selected(self):
        items_count = self.availableList.count()
        for i in range(items_count):
            item = self.availableList.takeItem(0)
            self.selectedList.addItem(item)

    def move_all_to_available(self):
        items_count = self.selectedList.count()
        for i in range(items_count):
            item = self.selectedList.takeItem(0)
            self.availableList.addItem(item)


class MainWindow(QMainWindow):
    """
    Main window class for the AWS Vulnerability Scanner application.

    This class sets up the UI and connects various signal-slot mechanisms for the application.
    """
    def __init__(self, app, parent=None):
        """
        Initializes the main window with layout and widgets.

        Args:
            app (QApplication): The application object.
            parent (QWidget, optional): The parent widget. Defaults to None.
        """
        super(MainWindow, self).__init__(parent)
        # Open the log file for writing
        try:
            # Open the log file for writing
            self.log_file = open("result_log.txt", "w")
        except Exception as e:
            print(f"Failed to open log file: {e}")
        self.central_widget = QWidget()
        self.setCentralWidget(self.central_widget)
        self.main_layout = QVBoxLayout(self.central_widget)
        self.app = app
        font = QFont("Arial", 12)
        app.setFont(font)

        # Apply Color Styling
        palette = QPalette()
        palette.setColor(QPalette.Window, QColor(53, 53, 53))
        palette.setColor(QPalette.WindowText, Qt.white)
        palette.setColor(QPalette.Base, QColor(15, 15, 15))
        palette.setColor(QPalette.AlternateBase, QColor(53, 53, 53))
        palette.setColor(QPalette.ToolTipBase, Qt.white)
        palette.setColor(QPalette.ToolTipText, Qt.white)
        palette.setColor(QPalette.Text, Qt.white)
        palette.setColor(QPalette.Button, QColor(53, 53, 53))
        palette.setColor(QPalette.ButtonText, Qt.white)
        palette.setColor(QPalette.BrightText, Qt.red)
        palette.setColor(QPalette.Highlight, QColor(142, 45, 197).lighter())
        palette.setColor(QPalette.HighlightedText, Qt.black)
        app.setPalette(palette)

        # Initialize the progressDialog here
        self.progressDialog = QProgressDialog("Scanning...", "Cancel", 0, 100, self)
        self.progressDialog.setAutoReset(True)

        # General window setup
        self.setWindowTitle('AWS Vulnerability Scanner and Dual List')
        self.setGeometry(100, 100, 800, 500)

        # Main Layout
        self.central_widget = QWidget(self)
        self.main_layout = QVBoxLayout(self.central_widget)
        self.setCentralWidget(self.central_widget)

        # Apply font size for result box
        result_font = QFont("Arial", 10)
        self.resultBox = QTextEdit(self.central_widget)
        self.resultBox.setFont(result_font)
        self.resultBox.setReadOnly(True)
        self.main_layout.addWidget(self.resultBox)


        # AWS Services Groups (defined once and reused)
        self.aws_services_groups = {
            "Compute": [
                    'EC2',
                    'Lambda',
                    'ECS',
                    'EC2_Auto_scaling',
                    'Elastic_beanstalk'
                ],

                "Storage": [
                    'S3',
                    'Elastic_transcoder'
                ],

                "Database": [
                    'RDS',
                    'DynamoDB',
                    'Redshift',
                    'Athena'
                ],

                "Networking": [
                    'VPC',
                    'Route_53',
                    'Cloudfront',
                    'ELB',
                    'API_gateway'
                ],

                "Security, Identity, & Compliance": [
                    'IAM',
                    'KMS',
                    'Secrets_Manager',
                    'WAF',
```

```
                'Inspector',
                'Cognito'
            ],

            "Analytics": [
                'Cloudwatch',
                'ElasticSearch',
                'GLUE'
            ],

            "Application Integration": [
                'SNS',
                'SQS',
                'Step_functions'
            ],

            "Deployment & Management": [
                'Cloudformation',
                'Codedeploy',
                'Cloudtrail'
            ],

            "Others": [
                'SWF',
                'ECR'
            ]
        }


        # Dual List setup
        # Add some spacing
        self.main_layout.setSpacing(15)
        self.setupAWServicesTree()  # Use the tree setup method
        self.setup_dual_list()  # Use the dual-list setup method

    def setupAWServicesTree(self):
        """
        Sets up a tree widget to display AWS services in categories.

        This method populates a QTreeWidget with AWS services organized by their categories.
        """
        self.tree = QTreeWidget(self)
        self.tree.setHeaderLabel("AWS Services")
        # Define color scheme for categories
        category_colors = {
            "Compute": QColor("blue"),
            "Storage": QColor("green"),
            "Database": QColor("red"),
            "Networking": QColor("cyan"),
            "Security, Identity, & Compliance": QColor("magenta"),
            "Analytics": QColor("yellow"),
            "Application Integration": QColor("orange"),
            "Deployment & Management": QColor("purple"),
            "Others": QColor("grey")
        }

        for category, services in self.aws_services_groups.items():
            parent = QTreeWidgetItem(self.tree)
            parent.setText(0, category)
            # Set the color for the category
            parent.setForeground(0, QBrush(category_colors.get(category, QColor("black"))))
            for service in services:
                child = QTreeWidgetItem(parent)
                child.setText(0, service)

        self.main_layout.addWidget(self.tree)


    def setup_dual_list(self):
        """ Sets up a dual list widget for selecting AWS services. """
            # Code to set up the dual list
        self.dualList = DualListWidget(self)

        formatted_services = []
        for group, services in self.aws_services_groups.items():
            formatted_services.append(f"== {group} ==")  # This is a group header, style it as needed
            formatted_services.extend(services)  # These are the individual services

        self.dualList.populate_available_services(formatted_services)
        self.main_layout.addWidget(self.dualList)

        # Button palette
        button_palette = QPalette()
        button_palette.setColor(QPalette.ButtonText, Qt.black)  # Setting the text color to black

        # Setup the Scan and Stop Scan buttons
        self.scanButton = QPushButton("Scan", self.central_widget)
        self.scanButton.setFixedSize(150, 40)  # Adjust button size
        self.scanButton.setPalette(button_palette)  # Set the palette to the button
        self.scanButton.clicked.connect(self.start_scan)

        self.stopScanButton = QPushButton("Stop Scan", self.central_widget)
        self.stopScanButton.setFixedSize(150, 40)  # Adjust button size
        self.stopScanButton.setPalette(button_palette)  # Set the palette to the button
        self.stopScanButton.clicked.connect(self.stop_scan)
        self.stopScanButton.setEnabled(False)

        # Create a horizontal layout for buttons
        self.button_layout = QHBoxLayout()
        self.button_layout.addWidget(self.scanButton)
        self.button_layout.addWidget(self.stopScanButton)
        self.button_layout.setAlignment(Qt.AlignCenter)
        self.main_layout.addLayout(self.button_layout)
    def start_scan(self):
        """ Initiates the scanning process for selected AWS services. """
        # (Function implementation...)
        try:
            selected_services = [self.dualList.selectedList.item(i).text() for i in range(self.dualList.selectedList.count())]

            # Create an instance of ScanThread and connect its signals
            self.scanThread = ScanThread(selected_services)
            self.scanThread.resultSignal.connect(self.append_to_result)
            self.scanThread.progressSignal.connect(self.update_progress)
            self.scanThread.finishedSignal.connect(self.scan_finished)
            self.scanThread.start()

            # Initialize and show the progress dialog
            self.progressDialog.setValue(0)
            self.progressDialog.show()

            self.stopScanButton.setEnabled(True)
            self.scanButton.setEnabled(False)  # Disable the scan button when the scan starts
            print("Scan started")  # Added for debugging
        except Exception as e:
            print(f"Error in start_scan: {e}")
    def update_progress(self, value):
        """ Updates the progress of the scanning process. """
        self.progressDialog.setValue(value)

    def stop_scan(self):
        """ Stops the ongoing scanning process. """
            if hasattr(self, 'scanThread'):
```

```
                self.scanThread._is_running = False
            self.progressDialog.cancel()

    def append_to_result(self, text):
        """ Appends the given text to the result box in the GUI. """
        self.resultBox.append(text)
        self.log_file.write(text + "\n")   # This line writes the text to the log file
        self.log_file.flush()   # This ensures that the content is actually written to the file and not just buffered

# Dummy scan functions for each AWS service


    def setup_aws_scanner(self):
        # Initialize AWS services or resources, if needed
        self.init_aws_resources()

        # Start the scanning process
        self.start_scan()

    def scan_finished(self):
        try:
            self.progressDialog.setValue(100)
            self.stopScanButton.setEnabled(False)
            self.scanButton.setEnabled(True)  # Re-enable the scan button when the scan is finished
            print("Scan finished")  # Added for debugging
        except Exception as e:
            print(f"Error in scan_finished: {e}")

    def closeEvent(self, event):
        if hasattr(self, 'log_file'):
            self.log_file.close()
        super(MainWindow, self).closeEvent(event)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main_window = MainWindow(app)  # Pass app as an argument
    main_window.show()
    sys.exit(app.exec_())
```