T1 Cybriant Attack Surface Management
Capstone Industry Partner

Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS 4850, Senior Project, Sec 01/03, Fall ‘24

Professor Perry

San Ciin, Nick Tanner, Nelson Thairu,

Danard McLemore, Keshaun Berry

11/17/2024

Website: Cybriant Attack Surface Management Team 1
GitHub: Cybriant Attack Surface Management Code

Num of Lines of Code	Num of Project Components/Tools	Total Man Hours
3626	15	481

Table of Contents

1.0 Introduction

This document serves as the comprehensive final report for the development of the Attack Surface Management System for Cybriant. This report consolidates all aspects, from initial specifications to the final design and implementation.

Our primary objective is to develop software that profiles the attack surface of small to mid-sized companies, aiming to improve their BitSight scores and reduce maintenance costs. The system scans digital environments to identify potential risks and vulnerabilities, providing actionable insights to enhance their security posture.

Utilizing the Google Cloud platform, the system integrates various security tools for a robust analysis of the digital environment's attack surface. The data is processed through a hybrid Linux and Python pipeline, transforming it into user-friendly visualizations in Looker Studio. This enables companies to clearly understand their security posture and address critical vulnerabilities effectively.

This document outlines the architecture, system design, and security considerations, ensuring the system is robust, scalable, and capable of effectively managing an organization's attack surface. It offers a detailed guide for developing a comprehensive solution that helps companies understand and strengthen their security posture.

2.0 Requirements

2.1 Design Constraints

2.1.1 Environment

This analytical software package will be part of a comprehensive security analysis suite of tools used to gain a better understanding of a company's security health. This package is designed to run in Google's Cloud environment. Python will be integrated into the cloud environment to facilitate the reconnaissance process, which involves gathering the data focused around various security metrics. The Python code will be stored and managed through GitHub. There will be automatic deployments from GitHub into the cloud environment.

The package will have two methods of invocation: one method will be an automatic invocation, and the other will be through a website. The results from the software will be stored into Google's BigQuery. A visualization component will then render the data in BigQuery. Optionally, the data will go through Google's Chronicle to structure the data in a more readable format.

2.1.2 User Characteristics

Security Analysts - These individuals should have knowledge of incident types and be able to recognize which type of log corresponds to each incident. Basic computer knowledge is required to navigate the dashboard visualization component of this software package. Basic CVE knowledge will be required to relay remedy information if any vulnerabilities are found in a client's configuration.

These individuals are responsible for interpreting the results shown in the dashboards. They are also responsible for creating detailed reports based on available information.

Clients - These individuals should have knowledge of basic security principles and practices. They should have familiarity with common security terms and concepts.

These individuals are responsible for utilizing the results to improve their security posture. They should also be proactive in seeking further improvements and maintaining their security health over time.

2.1.3 System

The analytical software package will require a few modifications to accommodate additional metrics. Any additional metrics can be run in their own Cloud Run job invocation. The master run job will need to be modified to invoke the additional metric. BigQuery will also need to be modified to accept the resulting data. If the resulting data type is not already present, a new table can be created to fulfill this purpose.

2.2 Functional Requirements

2.2.1 Google Cloud Service

Description

Google Cloud Service will hold a python script that updates and invokes the Google Cloud Run container. This service will act as the entry point to the entire pipeline.

Use Case: Entrance

- API POST call received from source.
- Python script gets domain passed from API call
- Script checks the tools last run time
- Script updates the Cloud Run containers environment variables
- Script invokes the image of the container corresponding with the updated tool

Requirements

- *REQ-1*: The API route must be known to invoke the service.
- *REQ-2*: The API route should be configured correctly in order to be invoked.

2.2.2 Google Cloud Run

Description

Google's Cloud Run will be used to hold and spawn instances of a similar Cloud Run Job. The Cloud Run Job will be configured to execute a different tool every time an instance is spawned. Every instance of the run job will be spawned from the script found in Google's Cloud Service.

Use Case: Modular Containers

- The Python script in the Cloud Service determines what tools to run
- The RUN command in the dockerfile is updated to match the DOMAIN parameter and the tool
- An instance of the container is spawned with the commands

Requirements

- *REQ-1*: The dependencies for every tool must be included in the dockerfile
- *REQ-2*: The exact command that corresponds to the tool to use must be known
- *REQ-3*: Some pre-existing knowledge of the used tool should be known to prevent errors

2.2.3 Python Data Parsing

Description

Python will be used to parse the results from the various security-related tools. The results from the tools will be transformed into a .csv file for integration with Google's BigQuery.

Use Case: Formatting

- Call the python script with the correct system arguments
- Extract the results from the security tool
- Transform the results into a format suitable to write to a .csv file
- Write the results to the .csv file

Requirements

- *REQ-1*: Provide an error status code and short description upon unsuccessful execution.
- *REQ-2*: Results from the security tool are to be expected.
- *REQ-3*: The output should be a .csv file for uploading into BQ

2.2.4 Google Cloud Storage

Description

Google’s Cloud Storage will hold two types of configuration files. One will be a ‘master file’ that holds links to every domain-specific configuration file. The other type will be the domain-specific configuration file itself. These files will be used to dynamically run a set amount of tools for every domain.

Use Case: Configuration Storage

- The ‘master’ configuration file is fetched from storage
- The domain specific configuration file is located in the master file
- The domain specific configuration file is fetched from the corresponding storage

Requirements

- *REQ-1*: Cloud Storage has been configured to store multiple files
- *REQ-2*: The master configuration file can hold functional links

2.2.5 GitHub Version Control

Description

GitHub will store the code, documentation, and configuration files as they are created and updated.

Use Case: Version Control

- Scripts are developed locally
- Comments are made on the changes since last upload
- Comments and changes are uploaded to the remote repository

Requirements

- *REQ-1*: The GitHub repository is configured to allow updates
- *REQ-2*: The repository allows for copying to a local machine
- *REQ-3*: The contributors are synced with the remote repository

2.2.6 Google BigQuery Storage

Description

Google’s BigQuery Storage will be utilized to aggregate any data gathered from the other tools. Both unstructured and structured data will be stored to have access to a client’s full profile.

Use Case: Storing

- Gather information specific to the client
- Format the data according to the schema in BigQuery
- POST the data to BigQuery

Requirements

- *REQ-1*: Provide an error status code and short description upon failure.
- *REQ-2*: The schema of the tables in BigQuery should be known.

2.2.7 Chronicle Integration (Optional)

Description

Google's Chronicle SIEM will be utilized to turn unstructured data into structured data. Chronicle uses Artificial Intelligence in order to accomplish this task.

Use Case: Structuring

- Gather pre-processed data from BigQuery.
- Format data to match the API call into Chronicle.
- POST the data into Chronicle and return the result.
- Format the result to align with BigQuery's schema.
- Store the result using a POST call to BigQuery.

Requirements

- *REQ-1*: Provide error codes upon failure.
- REQ-2: The format of the data that is returned from Chronicle should be anticipated.

2.2.8 Python Pre-Processing (Optional)

Description

Python will be utilized to process any data to be used in Chronicle. The data will be turned into a format that is suitable for Chronicle ingestion.

Use Case: Formatting

- Collect the data from BQ into a temporary variable.
- Format the data to be suitable for Chronicle.
- Send the data to the Chronicle.

Requirements

- *REQ-1*: The format of Chronicle's parser should be known..
- *REQ-2*: The ability to parse unstructured security logs will be required.

2.3 Non-Functional Requirements

2.3.1 Security Requirements

- Users should have access only to the specific client they are targeting.
- Users should not be able to gain any information beyond what is defined in the legal requirements.
- Users should not be able to gain access to any underlying systems.
- There should be software systems in place to prevent any operation from using an excessive amount of compute.

2.3.2 Legal Requirements

- No found vulnerabilities are to be exploited to gain access to a client's sensitive information.
- Clients' security configurations should not be used for malicious purposes.
- Clients should not have access to other clients' records.

2.3.3 Usability Requirements

- Functions should be labeled in a non-cryptic manner.
- Every function should be documented using standard comment practices.

2.3.4 Documentation and Training

User-guides are to be written to provide a step-by-step walkthrough of the program. Both a configuration perspective and user perspective are to be included. This guide will act as the software's documentation. The user guide will include examples of used parameters, as well as examples of the returned results.

2.4 External Requirements

2.4.1 User Interface Requirements

Curl Invocation

- The curl invocation method will consist of a URL that accepts either a single domain or a file that has a list of domains as a parameter.
- The parameters will be uploaded to the Cloud Service through a POST request.

Optional Dashboard interface

- Visualization software such as Looker Studio will be utilized to generate a visual overview of the found data for a client.
- The created dashboard will have various metrics that are relevant to the client.
- There will be supporting visuals, like graphs and charts, that go alongside the data.

2.4.2 Software Interface Requirements

- Use cases 1.2.1 will be available via RESTful API calls routed through the HTTPS protocol.
- Data storage will be defined in Google's BigQuery. The schema used is to be determined by the category the data pertains to.
- The pipeline will be invoked via a RESTful POST API call, which will be routed to the main Cloud Run job.
- The bulk structuring of the logs will be directed to Google's SIEM, Chronicle.
- The compute resources will be provided through Google's cloud services; Google Cloud Run.
- A front-end facing website will facilitate manual interaction with the pipeline.

2.4.3 Communication Interface Requirements

The following protocols will be used in this software:

- **HTTPS**: The Hypertext Transfer Protocol Secure is an extension of the HTTP protocol. Data is encrypted before it is sent to the server in this protocol.
- **DNS**: The Domain Name System translates IP addresses into human-readable names.
- **ARP**: The Address Resolution Protocol maps IP addresses to the physical address specific to the machine.
- **TLS/SSL**: The Transport Layer Security / Secure Sockets Layer protocols facilitate secure communication over the network. Verification must happen in order for any information to be transmitted over the network.
- **DNSSEC**: The Domain Name System Security Extensions protocol adds verification to any DNS response. This ensures that the returned query information has not been altered.
- **SPF**: The Sender Policy Framework protocol is used to authenticate emails by allowing domain owners to authorize specific mail servers.
- **DKIM**: The DomainKeys Identified Mail protocol uses signatures to verify that the sender of the mail is the actual sender and not someone else trying to impersonate them.
- **DMARC**: The Domain-based Message Authentication, Reporting and Conformance protocol is an extension of the DKIM and SPF protocols, which provide an even more secure mechanism for authentication. The additional mechanisms specify what to do during specific mail-based events.

3.0 Analysis

3.1 Feasibility

The proposed security metrics collection and analysis pipeline is highly feasible, leveraging cloud-native solutions that are well-supported and scalable. Google Cloud

Run and BigQuery are industry-standard tools that provide robust performance and flexibility. The choice of a containerized architecture with Docker further enhances the feasibility, allowing easy configuration, deployment, and maintenance of the environment. Python and Linux-based tools ensure compatibility and efficiency, as these technologies are widely used and well-documented. The use of GitHub for version control simplifies collaboration and updates, making the development process manageable. However, cost management and optimization must be considered, especially with large data volumes in BigQuery and high-frequency Cloud Run jobs.

3.2 Risk

Several risks are associated with this project. First, cost overruns are a significant concern, given the potential for high data ingestion and query expenses in BigQuery, as well as the cost of running numerous Cloud Run jobs. The project must implement strict cost-monitoring measures and optimize queries to mitigate this. Second, security risks exist because the system involves processing sensitive domain-related data, which could be a target for attackers. Proper access controls, secure API endpoints, and encryption must be enforced to safeguard data. Third, there is a risk of tool integration challenges. Each security tool has unique dependencies and output formats, and integrating new tools might lead to compatibility issues or data inconsistencies. Additionally, performance risks include potential inefficiencies when processing large volumes of data concurrently, necessitating continuous monitoring and optimization. Lastly, there is a risk of misconfigurations in cloud permissions, which could either compromise security or prevent tools from functioning correctly.

3.3 Requirement Prioritization

The project should prioritize core functionalities that are critical for delivering immediate value and ensuring system stability. The top priority should be setting up the core infrastructure, including Docker, Google Cloud Run, and BigQuery, as these are the backbone of the pipeline. Following this, the integration of essential security tools like Amass for subdomain enumeration and Nuclei for vulnerability scanning should come next, as these provide fundamental insights into domain security. Data parsing and schema enforcement should be prioritized to ensure data integrity and reliability in BigQuery. Security features, such as configuring access controls and secure API endpoints, should be a high priority to mitigate security risks. Finally, cost management tools and monitoring should be implemented to manage and optimize resource usage efficiently. Secondary priorities can include integrating additional tools, enhancing data visualization, and developing advanced machine learning models for deeper analysis.

3.4 Use

The pipeline is designed for continuous security assessment and monitoring of domains. Its primary use is to help organizations proactively identify vulnerabilities and security threats, such as open ports, domain squatting, and weak email authentication protocols. Security teams can leverage the pipeline for regular assessments, generating detailed reports and insights that inform remediation efforts. Additionally, the structured data in BigQuery can be used for further analysis, enabling the development of custom visualizations or machine learning models. This system can be extended to perform periodic checks, schedule automated scans, and alert teams to critical security issues. It can also serve as a valuable tool for penetration testers and cybersecurity analysts who need to gather and analyze large volumes of data efficiently.

3.5 Issues

Several potential issues could affect the successful implementation and operation of the pipeline. One major issue is tool output inconsistency. Security tools often have varying output formats, which can complicate data parsing and lead to integration challenges. Ensuring consistent and reliable parsing scripts will be critical. Data volume and query performance in BigQuery could also become an issue if not optimized, leading to slow query times or high costs. Managing concurrent Cloud Run jobs efficiently is another potential challenge, as running too many instances simultaneously could overwhelm resources or trigger rate limits in the cloud environment. Version control and updates might also pose a challenge, especially if tools frequently update or change their APIs, requiring the pipeline to be flexible and easily maintainable. Lastly, user error or misconfigurations in domain submissions and environment settings could disrupt the workflow, necessitating clear documentation and robust validation mechanisms.

4.0 Design

4.1 Goals

The primary objective of this project is to automate the collection, processing, and analysis of security metrics for domains using a containerized and scalable cloud environment. This is accomplished by leveraging Google Cloud Run and BigQuery for data processing and storage. The system will be capable of identifying vulnerabilities and threats in real-time, performing domain security checks efficiently, ensuring data is structured and formatted for effective storage and analysis, and supporting seamless integration of future tools.

4.2 System Design

The project architecture is built on cloud-native components and containerized environments, facilitating scalability, efficient data processing, and streamlined interactions. The main components include Google Cloud Run for managing the execution of containerized scripts for security analysis, Google BigQuery for storing and querying metrics, GitHub for version control and configuration file management, Docker for configuring the cloud environment using a custom Debian-based image, and a Python Flask API to serve as the interface for triggering data collection processes.

4.2.1 Constraints

The system must operate under constraints such as strict access controls for cloud environment permissions, efficient resource usage to minimize costs, and the ability to handle potentially large volumes of structured data in BigQuery. These constraints are crucial for ensuring security, cost-effectiveness, and scalability.

4.2.2 Composition

Core components of the system include Port Authority, which manages domain submissions via a web interface or command-line input, and triggering the Harbor Master job. Harbor Master manages domain configurations and invokes the Toolbox script to run security tools. Toolbox is responsible for executing these tools sequentially and processing their output, ensuring data is efficiently stored in BigQuery. A suite of parsers processes the output of tools like Amass, Nuclei, Subfinder, and Naabu to enforce schema consistency in BigQuery.

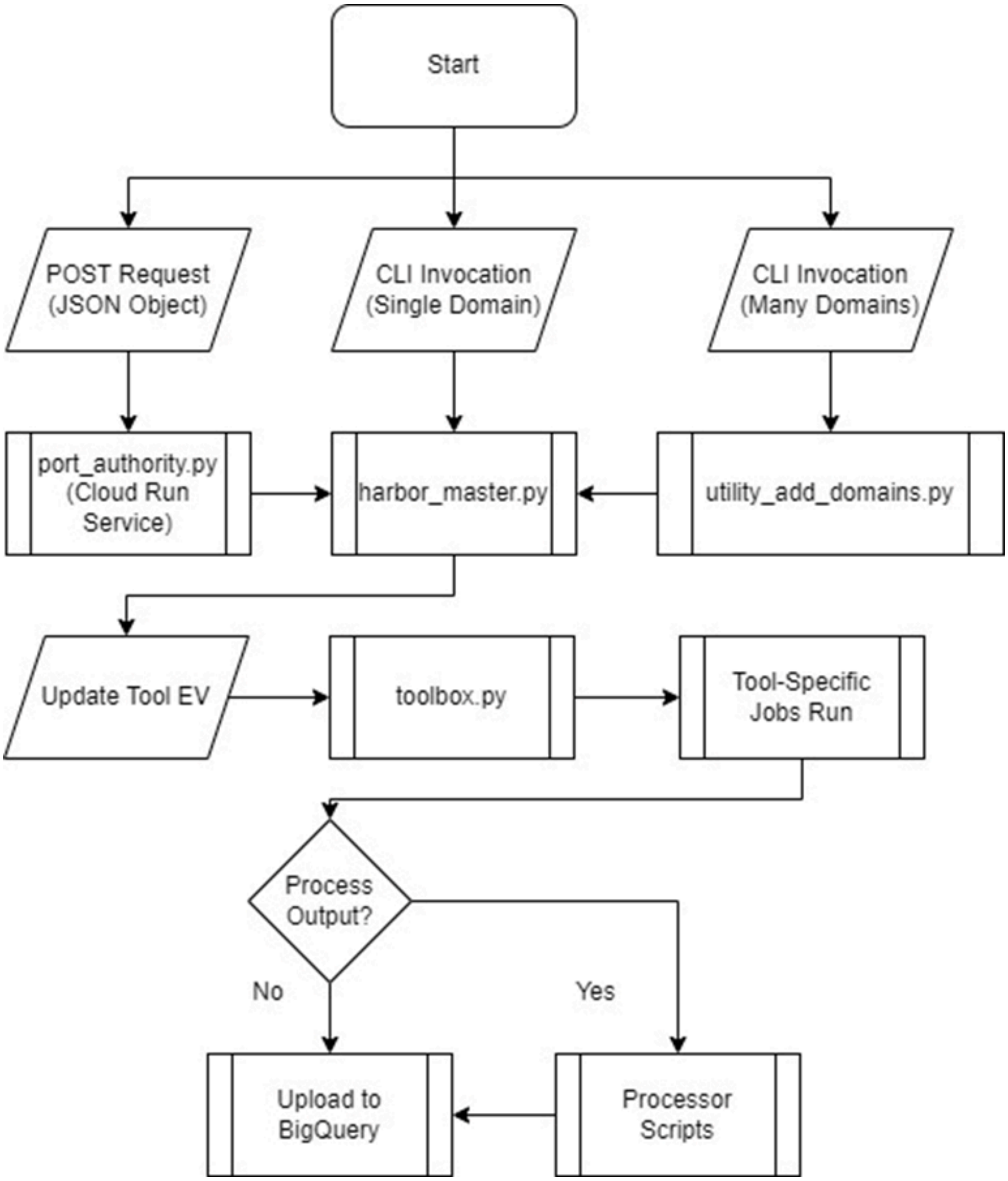


Fig 1. Overview of system's architecture

Security tools used include Amass for subdomain enumeration, Nuclei for vulnerability scanning, Subfinder for subdomain discovery, Httpx for HTTP probing, Katana for web crawling, Naabu for port scanning, DNSTwist for domain squatting detection, and custom scripts for TLS/SSL analysis, SPF/DKIM/DMARC checks, DNSSEC validation, and Punycode conversion.

4.3 Data Models

The system’s data model includes structured tables in BigQuery for each tool, with schemas ensuring data consistency. The master configuration file tracks domain configurations and the last run-time of tools, while domain configuration files contain tool settings, run intervals, and history for each domain. The data is structured to facilitate efficient querying and analysis.

4.4 Interfaces and APIs

The interface includes a Flask API that handles incoming POST requests for domain submissions and Cloud Run Jobs that execute containerized scripts to process domains. GitHub serves as the version control repository for all scripts, configuration files, and Docker setups. This modular and efficient design allows for easy updates and integration of new features.

4.5 Subsystems

4.5.1 Pipeline Management

Pipeline management consists of Port Authority, which acts as the entry point for domains, and Harbor Master, which manages configurations and invokes the Toolbox to run security tools. The Toolbox executes commands in a Linux environment, with error handling to ensure data is processed correctly.

4.5.2 Data Processing

Data processing involves custom parsers that structure tool outputs and store them as CSV files in BigQuery. Data handling includes taking JSON objects as input and outputting formatted CSV files. The system uses a modular design, making it easy to integrate new tools by updating configuration files and modifying Toolbox scripts.

4.5.3 Cloud Environment

The cloud environment setup uses a Dockerfile based on Debian, which installs necessary Python libraries and Linux tools. Dependencies are managed through a requirements file, making it straightforward to update and configure the environment. Google Cloud Run is used to execute jobs efficiently, scaling instances as needed.

4.6 Processing

Job execution begins with a domain submission through Port Authority, which triggers Harbor Master. Harbor Master checks for existing configurations or creates new ones, then invokes the Toolbox to run tools sequentially. The Toolbox ensures each command executes successfully, and parsers convert outputs into CSV files that are uploaded to BigQuery.

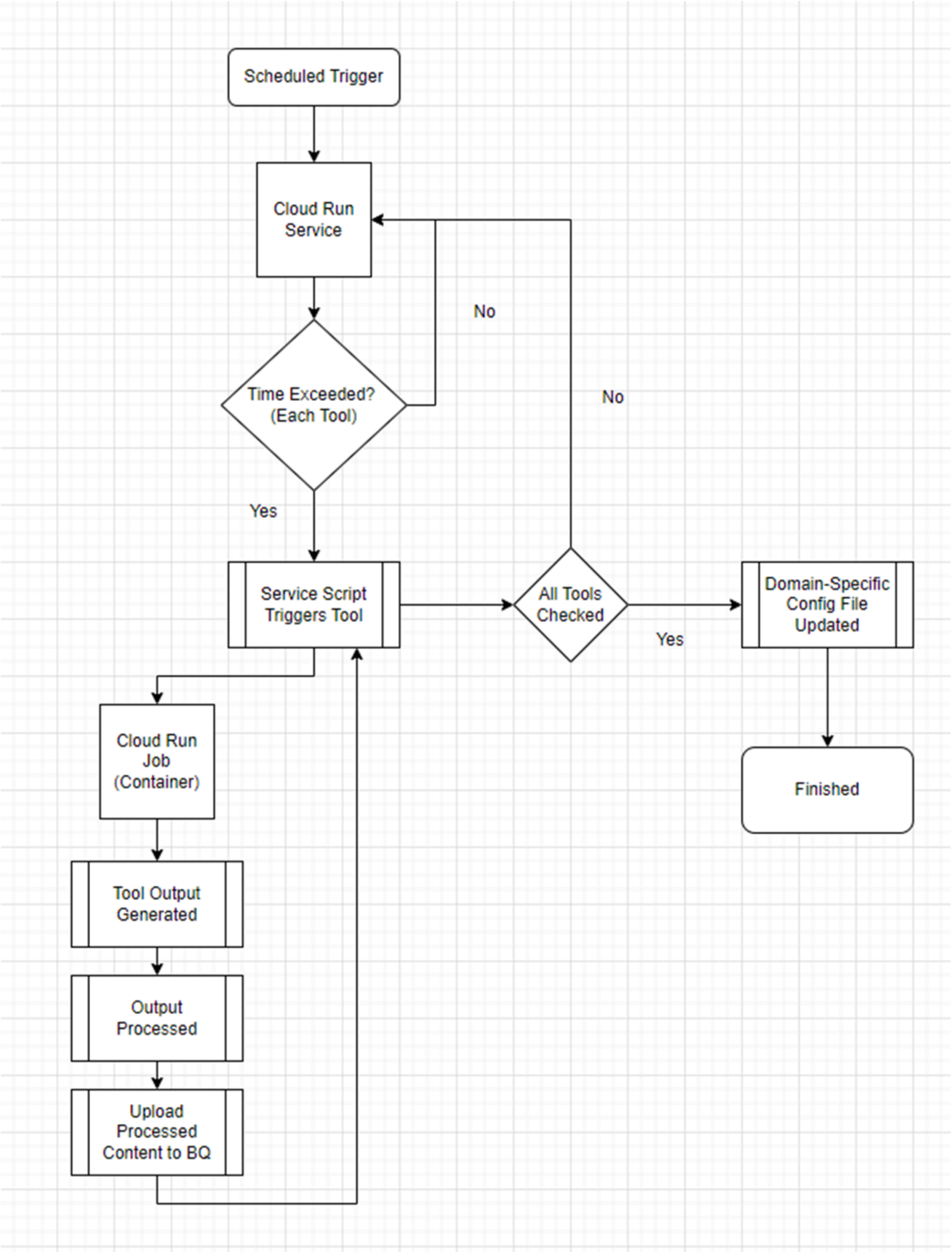

Scheduled Trigger
Cloud Run Service
No
Time Exceeded? (Each Tool)
No
Yes
Service Script Triggers Tool
All Tools Checked
Yes
Domain-Specific Config File Updated
Cloud Run Job (Container)
Finished
Tool Output Generated
Output Processed
Upload Processed Content to BQ

Fig 2. Outline of the harbor master script updating a domain's configuration

Data handling involves parsing and structuring tool outputs into CSV files that adhere to the defined schemas. The structured data is then uploaded to BigQuery, ensuring consistency and ease of querying for analysis and visualization.

4.7 Resources and Dependencies

Resources include Python libraries such as Flask, Nmap, idna, and urllib, and Linux tools like Amass, Nuclei, Subfinder, Naabu, and DNSTwist. The cloud services used include Google Cloud Run for job execution, BigQuery for data storage, and Google Cloud Storage for configuration file management.

4.8 Implementation

4.8.1 Dependencies

Dependencies are managed using a Dockerfile that defines the base Debian image, installs Python, and uses a requirements file to set up libraries. This setup ensures that the environment is fully prepared for executing security analysis tools.

4.8.2 BigQuery Setup

BigQuery is configured with table schemas that align with the metrics gathered from each tool. CSV files are parsed and uploaded to BigQuery, with schema enforcement ensuring data integrity and efficient querying.

4.8.3 Tool Integration

Integrating new tools is straightforward. Configuration files and Toolbox scripts are updated to include new tool commands, and the integration is tested to ensure compatibility. The system is designed to be flexible and support the seamless addition of new tools.

5.0 Development

5.1 Environment

5.1.1 Base configuration

Version Control

Our project utilizes GitHub to store updates to the code base, including the configuration needed for the cloud environment. The dependencies required by various tools are also stored in GitHub in a requirements text file, facilitating easier integration with the environment. This requirements file is used in the Dockerfile that configures the case service in Google Cloud.

Dockerfile Environment

Google’s cloud environment uses a Dockerfile to set the parameters, starting with identifying the base image for the operating system. We chose the Linux distribution, Debian, as our base. Debian provides a more comprehensive backbone for the environment and offers a richer selection of features due to the ‘apt’ package manager.

Environment Permissions

To run all the tools and install the dependencies, we had to configure some of Google’s permission in the cloud environment. For example, the ASM handler, a Google-provided tool used to configure our environment, needed to be assigned various admin roles to properly delegate the correct permissions to our Cloud Run Jobs. Once the appropriate service accounts had the correct permissions, we were able to configure the environment to implement our pipeline.

BigQuery environment

One component of our pipeline involved storing data in Google’s BigQuery. As previously stated, the service accounts needed various admin rights, including BigQuery. Once configured, BigQuery was set up according to the schema we created. Tables were created for each tool, with data formatted according to the table layout. The table schemas need to be referenced during the storing phase of the pipeline, so we have a reference script that holds each tool’s table schema.

Pipeline Environment

Entering the pipeline also had to be configured to enable invocation from outside the cloud environment. This was handled through a Python Flask route, allowing the environment to handle API requests. Once a port was configured, the environment was set to expose that route to the public internet. A POST request is made to our pipeline, passing in data as a JSON object, which is then parsed. The resulting data is passed onto the rest of the pipeline and handled accordingly.

The data gathered is based on factors recommended by our partner company, Cybriant. These metrics have associated tools best suited for them. The base Linux image we are using had some of these tools installed, while for others, the dependencies were identified and added to the Dockerfile that the cloud environment uses.

Image Configuration

Once all dependencies were resolved, we proceeded with packaging the configuration files to prepare them for import into the cloud environment. The configuration files were built through Docker and tagged for version control purposes. After the image was created, it was uploaded to the Artifact Registry in Google Cloud, where permissions played a crucial role. The cloud environment required authentication to accept the image, which was already taken care of.

5.1.2 Required Services

Cloud Run Service

Setting up Google's Cloud Run Service required the utilization of their default service account. The Run Service was configured to use a Python script as a base. This Python script accepts data from outside the environment, parses the data as a JSON object, and formats the configuration files. Once the configuration files are modified, the Cloud Run Service creates a Cloud Run Job that gathers metrics respective to the current configuration file.

Cloud Run Jobs

Every Cloud Run Job shares the same base configuration file. This is done to facilitate the environment which runs the scripts that gather the metrics. Each metric to be gathered has its own configuration that matches it. The tools that are chosen have an entire command sequence that handles gathering the data.

BigQuery

BigQuery accepts data for the various tools, with each tool having a table dedicated to it. The resulting data is imported into BigQuery in the same command sequence that the Cloud Run Job executes. This data is used to develop machine learning algorithms and visualizations for client use.

5.1.3 Expected Output

There are various methods that Google's BigQuery allows for the ingress of data. Our pipeline uses the CSV file method to upload data into BigQuery. We utilize custom Python scripts that parse the tool's data and turn it into CSV files. The formatting of the CSV files is created in a way that mimics the respective tool's table schema in BigQuery.

5.2 Components

5.2.1 Building the Pipeline

Port Authority and Harbor Master

There are two main ways to invoke the pipeline. Everything starts with either Port Authority or Harbor Master. Port authority serves as the entry point for any domains submitted via a web interface. It accepts domains through a JSON object via a POST request. Alternatively, domains can be processed directly using the command-line interface to invoke the Harbor Master script.

When a domain is submitted through Port Authority, it updates an environment variable for the Harbor Master job and triggers Harbor Master to run. Harbor Master then checks a master configuration file to see if the domain has been processed before. If the domain is new, it creates a domain-specific configuration file that includes a list of all tools to run, the last runtime, and an interval that can be set per tool per domain.

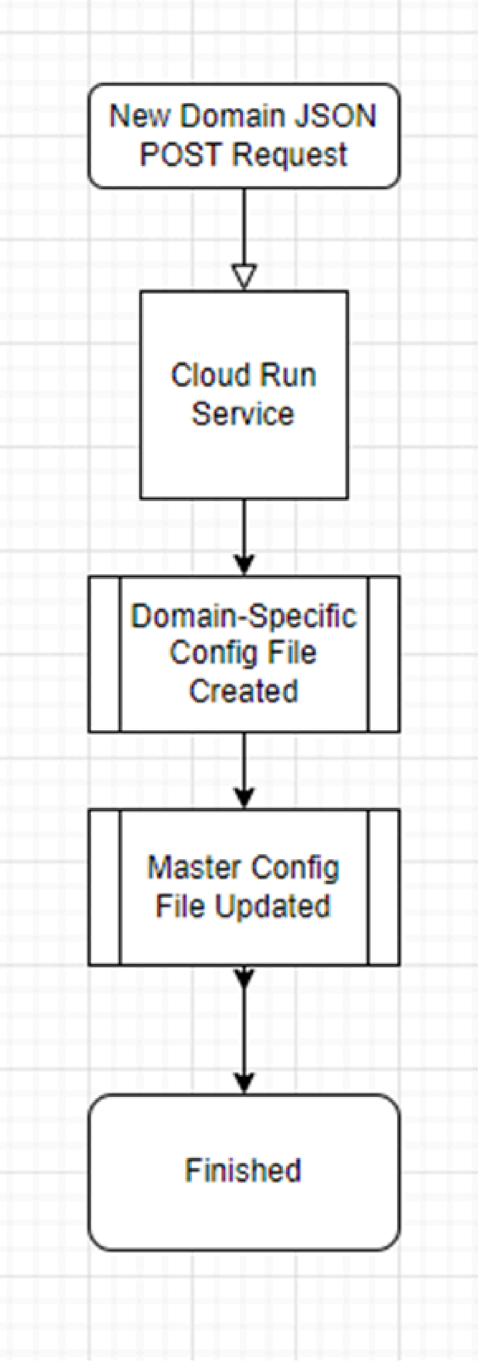

New Domain JSON POST Request
Cloud Run Service
Domain-Specific Config File Created
Master Config File Updated
Finished

Fig 3. Uploading a new domain to the pipeline.

These intervals can be adjusted to control how frequently each tool runs, allowing for flexibility in scheduling assessments. Users can update configurations using a utility script or by editing the files directly. Both Port Authority and Harbor Master update environment variables to invoke the Toolbox scripts and associated jobs.

Toolbox

Toolbox is responsible for executing all the tools against the domains. It operates within a container running Linux with Python scripts and uses string concatenation to construct command strings. Each command includes the tool execution, parser scripts for processing output, and instructions to create and load tables into BigQuery. The commands are concatenated using && to ensure sequential execution; if a command fails, subsequent commands do not run. Toolbox also handles tools that produce outputs requiring schema enforcement in BigQuery. For tools outputting CSV files, schemas are enforced to prevent BigQuery from misinterpreting header lines. This structured approach ensures that data is processed and loaded efficiently.

Toolbox includes several parsers that process tool outputs for further analysis and storage.

- **parser_amass_enum**: Parses enumeration results from Amass, extracting detailed information on discovered subdomains, associated IP addresses, and data sources. This structured data supports comprehensive subdomain analysis and asset discovery.
- **parser_amass_intel**: Processes Amass Intel's output, extracting intelligence data related to domain infrastructure, including organization details, IP ranges, and ASN information. This data is formatted for efficient querying and strategic insights.
- **parser_PD_httpx**: Extracts and formats HTTP response details, including status codes, headers, and response times from the output of the HTTPX tool. The parsed data is structured in a CSV file for efficient querying and analysis in BigQuery.
- **parser_PD_katana**: Processes the output from Katana, a web crawler, extracting discovered URLs, link information, and crawling metadata. It organizes data to support link analysis and asset discovery.
- **parser_PD_naabu**: Parses Naabu's port scanning results, extracting details like open ports, IP addresses, and scan timestamps. The data is formatted to a CSV file.
- **parser_PD_nuclei**: Handles Nuclei’s vulnerability scanning results, extracting information on detected vulnerabilities, severity levels, and affected assets. It ensures that the parsed data is easy to query and link to remediation efforts.
- **parser_PD_subfinder**: Extracts subdomain information from Subfinder's output, structuring the data to facilitate domain and subdomain enumeration.

Security Metrics Collection

Our security assessment focuses on several key areas, starting with email authentication protocols: SPF, DKIM, and DMARC. These protocols authenticate emails and prevent spam. For SPF, we check the domain's DNS TXT records for SPF entries. DKIM involves looking for specific TXT records indicating DKIM is enabled, while DMARC searches for DMARC TXT records associated with the domain.

DNSSEC enhances DNS security by adding cryptographic signatures. We check for the DNSKEY record in the domain's DNS records, and the presence of this record indicates DNSSEC is enabled. TLS/SSL certificates are also analyzed to verify website ownership and encrypt web traffic. We establish a secure connection to the domain and use commands to retrieve certificate details, including the TLS version and certificate information.

Punycode conversion is another crucial component of our analysis. It handles internationalized domain names by converting Unicode to ASCII. This is useful for domains with non-ASCII characters, as it allows us to detect domain variations effectively. For instance, a domain like 例え.com would be converted to xn--r8jz45g.com.

We also focus on detecting domain squatting threats using DNS Twist. This tool generates domain permutations to identify registered domains that closely resemble the target domain. The process helps detect typo-squatting and other forms of domain squatting, providing data on active threats. For example, DNS Twist can generate thousands of permutations and identify which ones are actively registered, highlighting potential risks.

Data Storage and Processing

All collected security metrics are stored in CSV files, which are then uploaded to Google BigQuery. The data is structured for efficient querying and analysis, allowing us to quickly identify and address security vulnerabilities.

Port Scanner

The Port Scanner tool identifies vulnerabilities associated with open ports on a domain. Open ports can expose services to exploitation, making it essential to assess the security posture of a domain. We wrote a port scanner using Nmap. Nmap is a versatile tool that provides detailed information, including whether the host is up and the open ports.

Our implementation uses the Nmap Python library to run port scans efficiently. The function accepts a text file containing a list of domains or a single domain and performs a stealth scan to avoid detection by security measures. We use multithreading to speed up the scanning process, collecting data points such as domain name, host IP, root domain, host state, and open ports. The results are written to a timestamped CSV file and uploaded to Google BigQuery for storage and analysis.

To handle domains with security measures in place, we employ stealth scanning techniques. Additionally, Nmap can perform partial connections to gather data without completing a full handshake, helping us avoid detection and blocking. Our system collects and organizes data points into lists, which are then stored in CSV format and uploaded to BigQuery. The structured data is easy to query, making it suitable for weekly security assessments.

5.2.2 Adding Tools to the Pipeline

Integrating new tools into the pipeline is straightforward. We develop or adapt the tool to work in our containerized Linux environment, usually using Python. Each tool is containerized, and a function is created to handle its execution and output parsing. The tool's output is structured for seamless integration with our parsers and BigQuery.

Configuration files managed by Harbor Master are updated to include the new tool, with specified execution parameters and intervals. Toolbox scripts are modified to include the tool in the command sequence. Any necessary environment variables or settings are added, and the integration is thoroughly tested to ensure compatibility.

5.2.3 Google Cloud Run Service

Google Cloud Run manages job execution, providing scalability and efficiency. We configure each tool or toolset as a Cloud Run Job, specifying container images, commands, and resource allocations. Jobs dynamically accept input data, like domain lists, and spawn instances in response to requests or on a schedule. This setup allows parallel processing and optimal resource use.

Command sequences are executed in order, with error handling to prevent cascading failures. Logs are generated for performance monitoring and troubleshooting. This scalable infrastructure efficiently processes large data volumes.

5.2.4 Expected Output for the Tools

Each tool generates specific outputs:

- **Email Authentication**: Records for SPF, DKIM, and DMARC configuration.
- **DNSSEC**: Verification of cryptographic signatures.
- **TLS/SSL**: Certificate issuer, validity, and supported TLS version.

- **Punycode Conversion**: ASCII versions of internationalized domains.
- **Domain Squatting Detection**: Registered domain permutations.
- **Port Scanning**: Open ports, host status, and subdomains.

Raw data from each tool is parsed and structured for BigQuery. We enforce schema consistency to ensure seamless integration, avoiding issues like auto-generated field names. Parsed data is stored in CSV or JSON format, compatible with BigQuery's requirements.

5.2.5 Component Interactions

Components communicate seamlessly to ensure smooth data flow. Port Authority handles domain submissions and sets variables for Harbor Master. Harbor Master manages configurations and invokes Toolbox scripts, which run tools in sequence. Google Cloud Run manages job execution, scaling instances for efficient processing.

Commands are executed in a defined order, with error handling to prevent failures from propagating. Data from all tools is formatted and uploaded to BigQuery for analysis, with visualizations created using Looker Studio.

5.3 Implementation

5.3.1 Implementing Dependencies

Every component that is part of the pipeline has dependencies. These dependencies are specific to the methods that the component uses. The components that use specific Python libraries need to be correctly installed and set up. The same goes for any components that use Linux commands that are not installed by default in the Debian distribution.

Luckily, it is relatively easy to correctly configure these dependencies. The only thing required for our pipeline to download and install these dependencies is the name of the library or tool to be utilized. Due to the backbone of the pipeline being a Dockerfile, we are able to install necessary dependencies during the creation of the environment.

The Dockerfile gets the requirements for the Python environment from a separate requirements file. This file holds every library that is currently utilized. New libraries can be added by appending them to this requirements file. The Linux tools that are used are installed beforehand.

During the creation of the environment, the Dockerfile updates and installs the base Linux packages. The Linux version of Python is also installed during this step, and the aforementioned requirements file is used to install the respective Python libraries. Any other base tools, such as the Go interpreter, are also configured to download and install in the working environment. Once these dependencies are configured, our project's code is copied into the working environment's directory and set to execute the cloud environment.

5.3.2 Implementing BigQuery

The schema for the tables is developed based on the results of each security metric from the respective tool. This ensures that only light editing is needed to capture the most important data. The schema is required to store the data into BigQuery. The schema is stored in a settings file that is called on and parsed. When the settings file is parsed, the part of the command sequence that handles storing the data is modified according to the tool, domain, and data being used.

5.3.3 Setting up the Tools

5.3.3.1 Cloud Run Service

The following tools were developed and implemented as components in the Google Cloud Service aspect of our pipeline:

port_authority.py

The port_authority script handles the POST requests coming in from outside the environment. The following functions are present in this script:

- **update_job_env_variable:** This function updates the environment variable for the Google Cloud Run job to reflect the domain. The domain is used as a parameter to call the harbor_master script.
- **trigger_container_job:** This function triggers the execution of the Google Cloud Run job. The constant script variables are used to invoke the pipeline
- **trigger_job:** This function triggers a job based on a POST request containing a domain name. The domain is taken from the JSON formatted data from the incoming request.

harbor_master.py

The harbor_master script fetches the configuration files for every domain whose metrics are being checked. The following functions are present in this script:

- **update_master_config_safe:** This function updates the master config for a domain in a thread-safe manner. The domain is used as a parameter in the update_master_config function.
- **read_domains_from_file:** This function obtains a list of domains from a .csv, .txt, or .json file. The file_path is checked for the file type and the data is gathered according to the file type.
- **execute_toolbox:** This function executes a tool job for a specific domain in the cloud, updating the progress bar. The domain, tool, and silent parameters are used to update the environment variables and execute the Cloud Run jobs. The last run-time of the domain-tool pairings are updated.
- **process_domain:** This function processes a single domain with error handling and silent mode. The domain is used in the configuration file and checks what tools should run. The silent bool determines whether to run the commands silently. The master configuration list is queried to see if the domain respective configuration file already exists.
- **process_domains_threaded:** This function processes each domain in a new thread with lock-based control for each tool, displaying progress bars. The domain is put into a queue and handed out to the available threads. Each thread invokes the process_domain() function with the given domain.
- **process_domains_serially:** This function processes each domain serially, updating configurations and running tools without threading. The list of domains are iterated through one by one passed as a parameter in the calling of every tool.

config.py

This script stores the general configuration settings for the cloud environment. Other constant parameters are also stored in this such as; the cloud storage bucket name, the master configuration file name, the domain specific configuration file name, and some parameters that are used in the BigQuery schema.

tool_settings.py

This script stores parts of the command sequence. The parts are stored as a dictionary, so the values are queried based on the key value. This allows for a modular approach to executing every tool.

utility_config_utils.py

This script controls the spawning of every Cloud Run Job instance. The following functions are present in this script:

- **Download_master_config:** This function downloads the master configuration file from Google Cloud Storage. Google Cloud storage is connected to and the master configuration file is downloaded by its file name.
- **Update_master_config:** This function adds a new domain to the master configuration file locally. The master configuration file is checked to see if the domain exists. If the domain doesn't exist, a link leading to the domain configuration file is written to the master file.
- **Upload_master_config:** This function uploads the updated master configuration file to Google Cloud Storage. Google Cloud storage is connected to and the master file is uploaded to it.
- **Load_domains_from_config:** This function loads the list of domains from the master configuration file. The master file is read and the specific domain is fetched from it.
- **Check_domain_in_master:** This function checks if a domain exists in the master configuration.
- **Download_domain_config:** This function downloads the domain-specific configuration file from Google Cloud Storage. Google Cloud storage is connected to and the domain is used to download the respective configuration file.
- **Create_domain_config:** This function creates a domain-specific configuration file and uploads it to Google Cloud Storage. A JSON file is created to hold details about the run-time of the tools against that domain.
- **Upload_domain_config:** This function uploads the domain-specific configuration file to Google Cloud Storage. Google Cloud storage is connected to and the domain specific file is uploaded to it.
- **Check_tools_to_run:** This function checks which tools need to be run for a domain. The last run-time of the tools are compared to the run intervals. If the last run-time is longer than the interval, that tool is run against the domain.
- **Update_last_run_time:** This function updates the last run time for a specific tool in the domain's configuration.
- **update_job_env_variable:** This function updates the environment variable for a specific job in Google Cloud Run. The toolbox script is called using the domain and tool to update the environment variables for that job instance.
- **Run_cloud_job:** This function runs a job in Google Cloud Run.

5.3.3.2 Cloud Run Jobs

The following tools were developed or utilized and implemented as components in the Google Cloud Run Jobs aspect of our pipeline:

custom_domain_security

This script queries the DNS table of a domain to gather records relevant to the required metrics. The following functions are present in this script:

- **get_spf_record:** This function queries the DNS table for the TXT record. The domain of the target is taken in as a parameter and the DNS table is queried. If the TXT record is found, it filters out the SPF record and returns that.
- **get_dkim_record:** This function queries the DNS table for the TXT record. The domain of the target is taken in as a parameter and the DNS table is queried. If the TXT record is found, it filters out the DKIM record and returns that.
- **get_dmarc_record:** This function queries the DNS table for the TXT record. The domain of the target is taken in as a parameter and the DNS table is queried. If the TXt record is found, it filters out the DMARC record and returns that.
- **get_tls_details:** This function fetches the TLS details for the given domain. The domain of the target is taken in as a parameter and a socket is created that connects to the domain and fetches its TLS details. Once the socket has been created, the TLS certificates are queried.
- **get_dnssec_status:** This function queries the DNS table for the DNSSEC records. The domain of the target is taken in as a parameter and the DNS table is queried. If the DNSKEY record is found, it checks if the resource record set of the response is present.
- **convert_to_punycode:** This function converts the given domain to the Punycode format. The domain of the target is taken in as a parameter and converted to punycode format. The idna library is used for this task.
- **gather_metrics:** This function gathers various security metrics for a domain. The domain is used as a parameter when calling every respective metric gathering function.
- **write_to_csv:** This function writes data to a CSV file.
- **process_domain:** This function processes a single domain and writes metrics to the specified output file. The domain is passed as a parameter to the gather_metrics function. The output path is passed as a parameter to the write_to_csv function.
- **format_output_path:** This function formats the output file path based on the domain name and the provided output path.

custom_port_scanner

This script scans a domain for any open ports. The following functions are present in this script:

- **get_root_domain:** This function extracts the root domain from a given domain. The domain is passed to the urlparse function of the urllib.parse package. The network location of the domain is returned.
- **scan_ports:** Scans the specified domain for open ports using nmap.
- **save_to_csv:** This function saves scan results to a CSV file in the specified output file path. The output file is opened for writing the data to. The results are parsed and the data is loaded into the CSV file.
- **process_domain:** This function processes a single domain by scanning ports and returns the scan results.
- **format_output_path:** This function formats the output file path based on the domain name and the provided output path.

custom_squatter_security

This script checks for squatting threats against a domain. The following functions are present in this script:

- **extract_squat_info:** his function extracts domain names and squatting types from squatting threat details. The squatters List is used in a list comprehension to extract the squatter domains and types.
- **squatter_summary:** This function summarizes information on detected squatting threats and writes details to a CSV file. The data about the squatters are extracted through the extract squat info function and passed as parameters to the csv creation function.
- **check_domain_squatting:** This function checks for domain squatting threats using the `dnstwist` tool. Permutations of the passed domain names are generated, identifying any registered variations. The output file is used as a parameter when the squatter summary function is called.
- **process_domain:** This function processes a single domain by checking for squatting threats and writing results to a CSV. This function is a target for multi-threading purposes that passes the domain and output file to the check domain squatting function.
- **format_output_path:** This function formats the output file path based on the domain name and the provided output path. The output path is checked to see if it is a directory, if it is not it is further checked to see if it is a CSV file. The domain is injected into the name of the file.

amass

This tool is used against a domain to find all of the sub-domains tied to it. The following functions are present in this script:

- **process_amass_enum_output:** This function processes the Amass enum output and saves it to a CSV file. The results this function uses comes from the execution of the tool in the Linux command line.
- **parse_amass_intel:** This function parses the Amass Intel output file and saves it to a CSV.The results this function uses comes from the execution of the tool in the Linux command line.

Project Discovery tools

These suites of tools are used to gain a comprehensive overview of a domain. The metrics that are gained from these suites include vulnerability scanning, HTTP probing, and web crawling. These tools have the following functions to manage the output they create:

- **Nuclei** - **parse_nuclei_output:** This function parses the Nuclei output and saves it to a CSV file.
- **Subfinder** - **parse_subfinder_output:** This function parses the Subfinder output and saves it to a CSV file.
- **Httpx** - **clean_csv:** This function cleans the input CSV file by ensuring no duplicate column names and removing empty columns.
- **Katana** - **parse_katana_output:** This function parses the Katana output and saves it into a CSV file.
- **Naabu** - **clean_csv:** This function cleans the CSV file by removing all header rows and saves the cleaned data.

6.0 Testing

6.1 Strategies

To ensure the robustness and reliability of our Attack Surface Management System, we employed several testing strategies.

6.1.1 Unit Testing

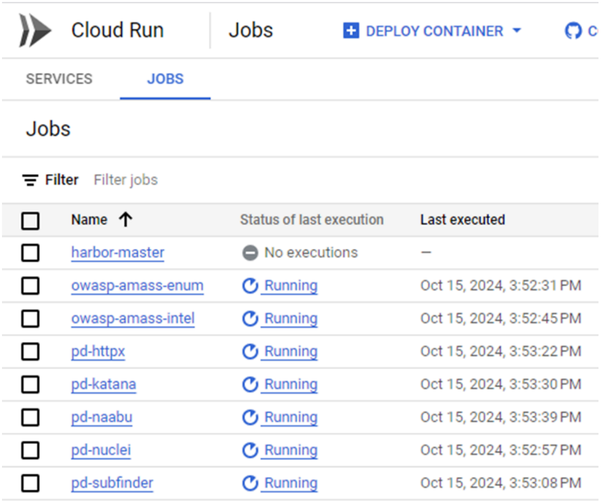
Unit testing was conducted on individual functions to verify their correctness and reliability in isolation.

6.1.2 Integration Testing

Integration testing was performed to ensure that different components of the system work together seamlessly when integrated into the pipeline.

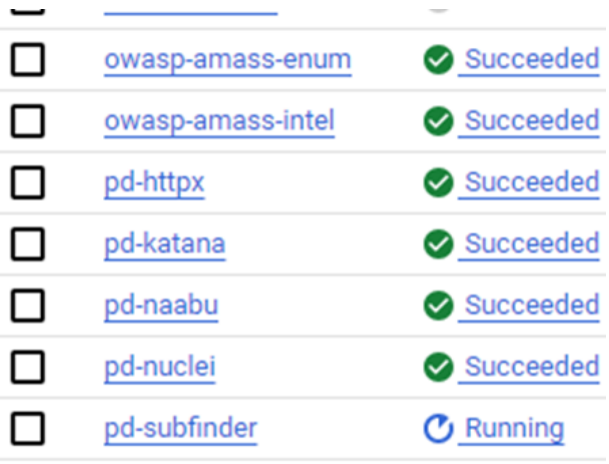
6.1.3 System Testing

Finally, system testing was carried out on the overall cloud architecture. This included iterative testing for correct permissions and the functionality of services such as BigQuery, Cloud Run, and Cloud Storage, ensuring that all components interacted as expected in a real-world environment.



□	Name ↑	Status of last execution	Last executed
□	harbor-master	No executions	–
□	owasp-amass-enum	Running	Oct 15, 2024, 3:52:31 PM
□	owasp-amass-intel	Running	Oct 15, 2024, 3:52:45 PM
□	pd-httpx	Running	Oct 15, 2024, 3:53:22 PM
□	pd-katana	Running	Oct 15, 2024, 3:53:30 PM
□	pd-naabu	Running	Oct 15, 2024, 3:53:39 PM
□	pd-nuclei	Running	Oct 15, 2024, 3:52:57 PM
□	pd-subfinder	Running	Oct 15, 2024, 3:53:08 PM

Fig 4. Testing the cloud environment

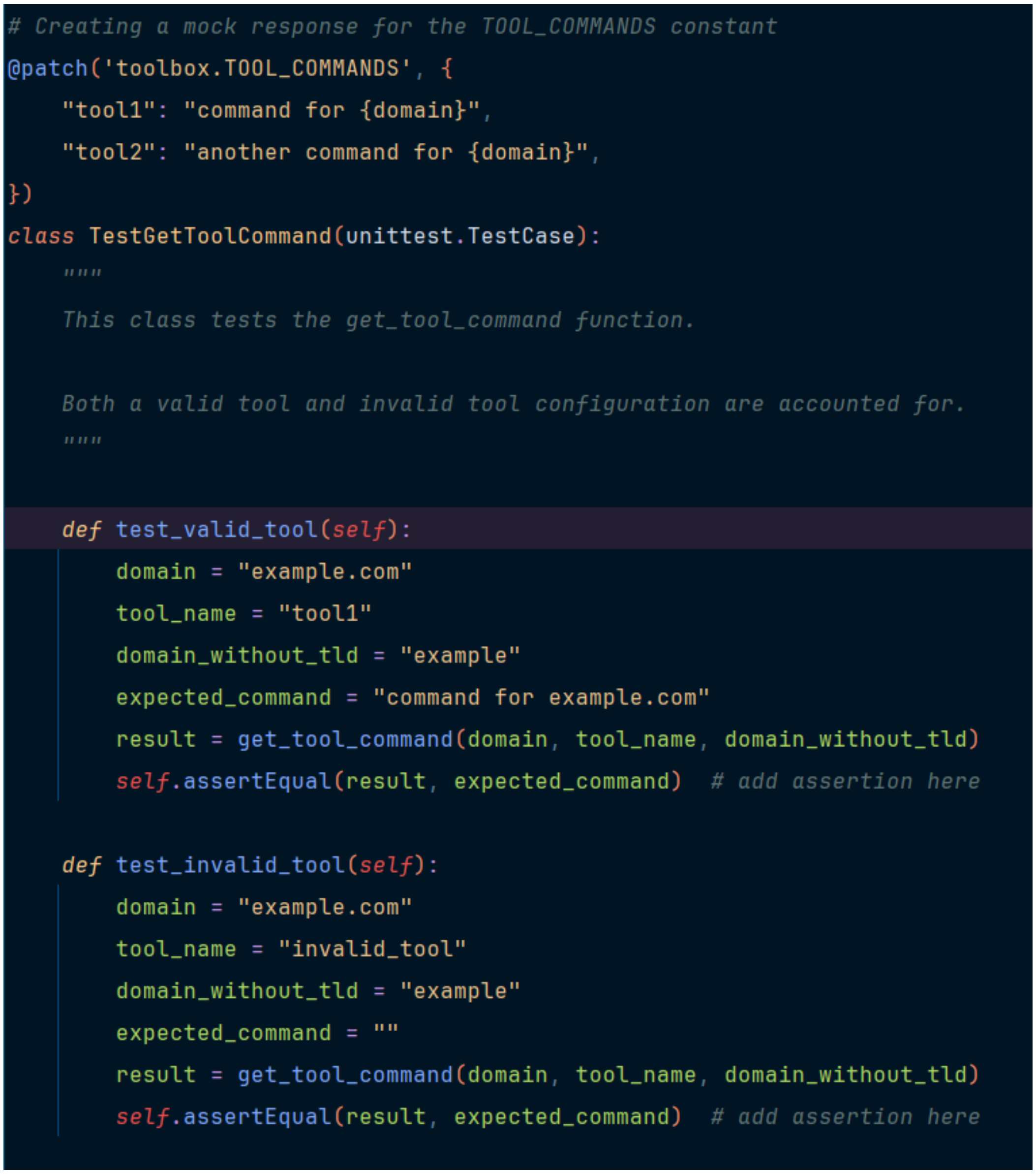


□	owasp-amass-enum	✓ Succeeded
□	owasp-amass-intel	✓ Succeeded
□	pd-httpx	✓ Succeeded
□	pd-katana	✓ Succeeded
□	pd-naabu	✓ Succeeded
□	pd-nuclei	✓ Succeeded
□	pd-subfinder	Running

Fig 5. Results of the systems test

6.2 Test Cases

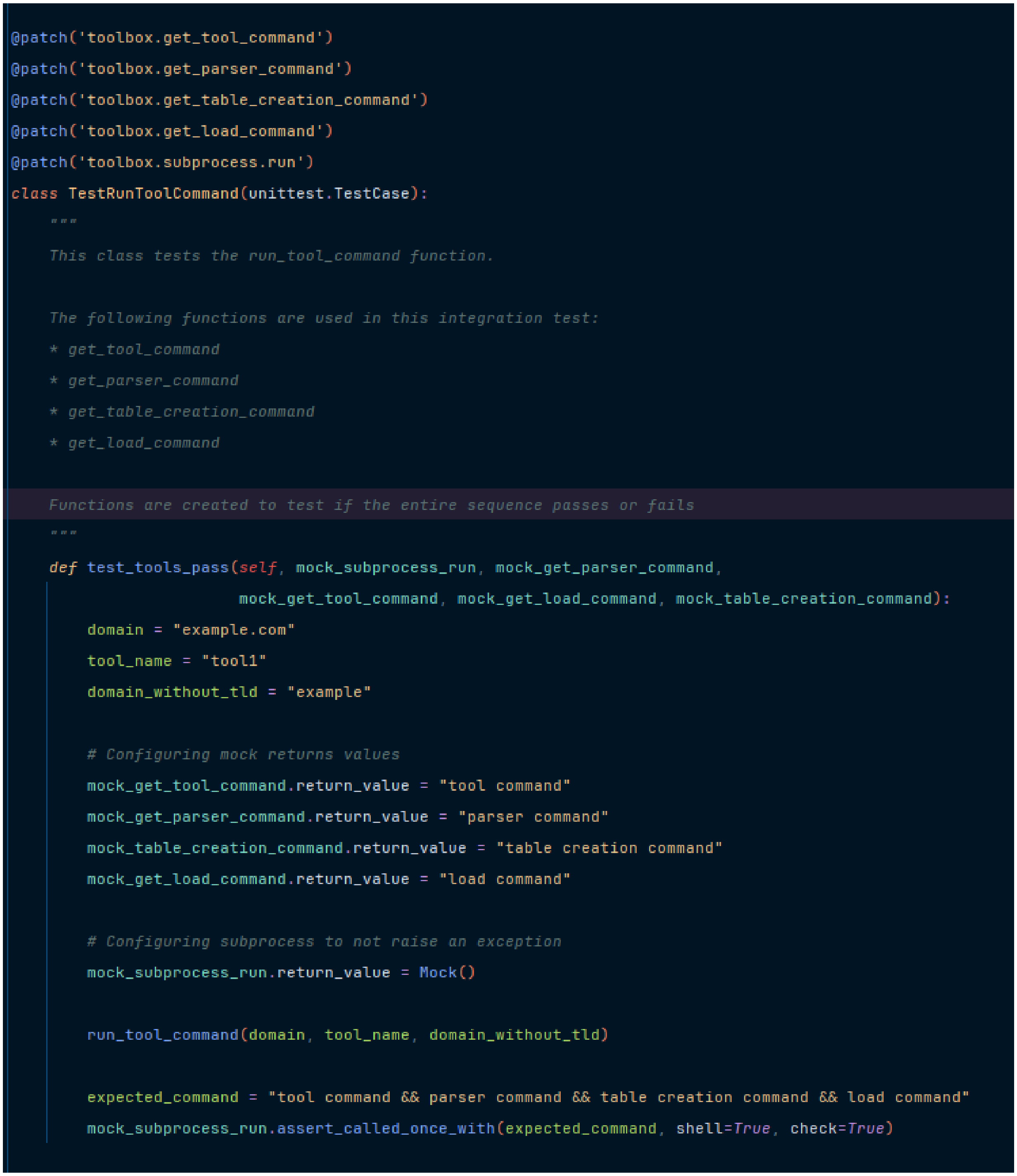
Specific test cases were designed to cover critical functionalities of the system. One test case focused on adding new components (tools) to the cloud environment, which required only the naming of dependencies in the configuration file and specifying the command sequence.



```python
# Creating a mock response for the TOOL_COMMANDS constant
@patch('toolbox.TOOL_COMMANDS', {
    "tool1": "command for {domain}",
    "tool2": "another command for {domain}",
})
class TestGetToolCommand(unittest.TestCase):
    """
    This class tests the get_tool_command function.

    Both a valid tool and invalid tool configuration are accounted for.
    """

    def test_valid_tool(self):
        domain = "example.com"
        tool_name = "tool1"
        domain_without_tld = "example"
        expected_command = "command for example.com"
        result = get_tool_command(domain, tool_name, domain_without_tld)
        self.assertEqual(result, expected_command)  # add assertion here

    def test_invalid_tool(self):
        domain = "example.com"
        tool_name = "invalid_tool"
        domain_without_tld = "example"
        expected_command = ""
        result = get_tool_command(domain, tool_name, domain_without_tld)
        self.assertEqual(result, expected_command)  # add assertion here
```

Fig 6. Class that handles the testing of various tools

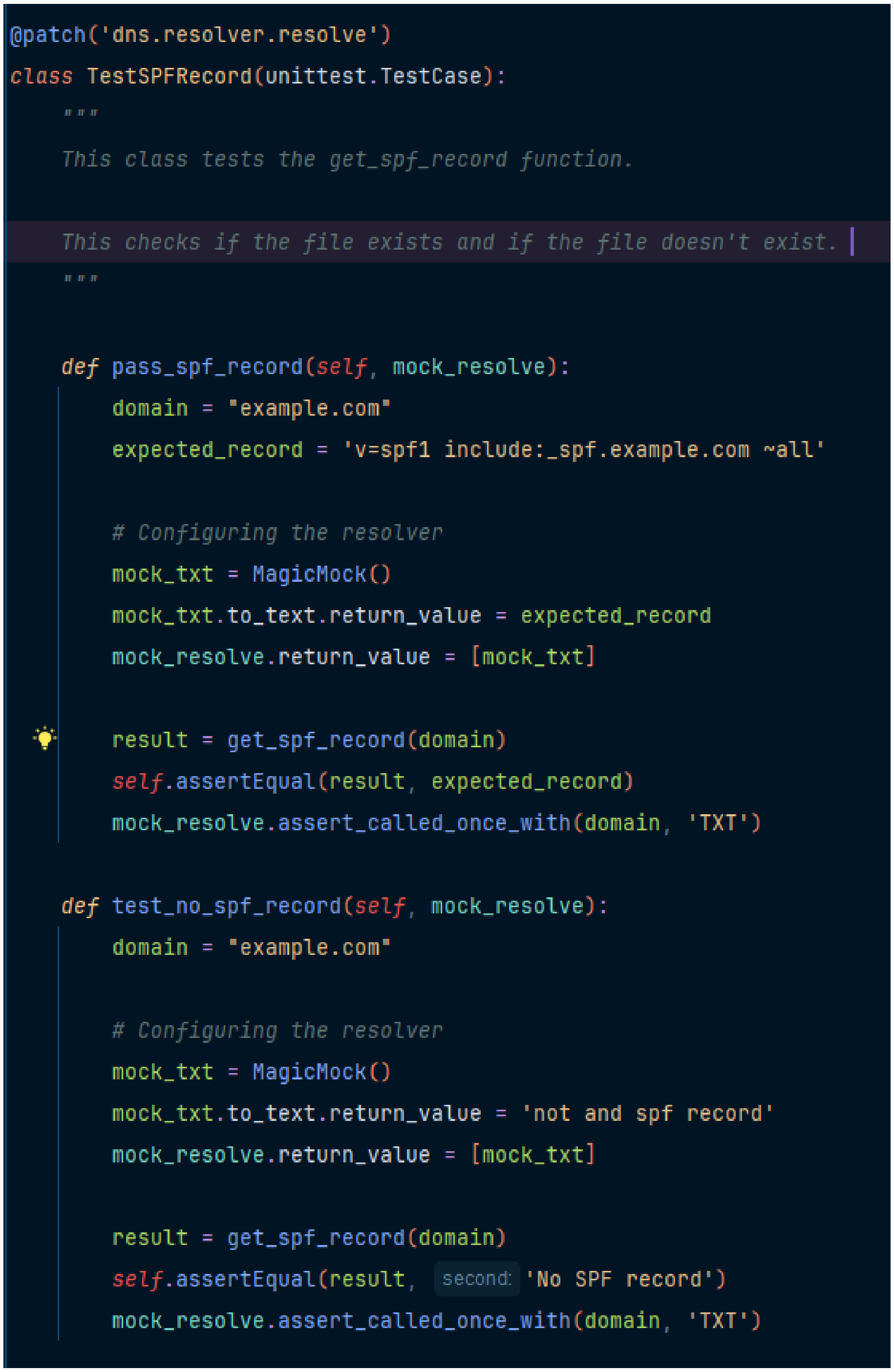
Another test case involved uploading the results to BigQuery, which was handled as part of the command sequence, ensuring that all the data reached the intended destination without errors.



```python
@patch('toolbox.get_tool_command')
@patch('toolbox.get_parser_command')
@patch('toolbox.get_table_creation_command')
@patch('toolbox.get_load_command')
@patch('toolbox.subprocess.run')
class TestRunToolCommand(unittest.TestCase):
    """
    This class tests the run_tool_command function.

    The following functions are used in this integration test:
    * get_tool_command
    * get_parser_command
    * get_table_creation_command
    * get_load_command

    Functions are created to test if the entire sequence passes or fails
    """
    def test_tools_pass(self, mock_subprocess_run, mock_get_parser_command,
                        mock_get_tool_command, mock_get_load_command, mock_table_creation_command):
        domain = "example.com"
        tool_name = "tool1"
        domain_without_tld = "example"

        # Configuring mock returns values
        mock_get_tool_command.return_value = "tool command"
        mock_get_parser_command.return_value = "parser command"
        mock_table_creation_command.return_value = "table creation command"
        mock_get_load_command.return_value = "load command"

        # Configuring subprocess to not raise an exception
        mock_subprocess_run.return_value = Mock()

        run_tool_command(domain, tool_name, domain_without_tld)

        expected_command = "tool command && parser command && table creation command && load command"
        mock_subprocess_run.assert_called_once_with(expected_command, shell=True, check=True)
```

Fig 7. Integration test that strings together commands to upload to BigQuery

Additionally, we tested the successful gathering of metrics using reconnaissance tools, which involved parsing the gathered data into CSV files for further analysis and storage.



```
@patch('dns.resolver.resolve')
class TestSPFRecord(unittest.TestCase):
    """
    This class tests the get_spf_record function.

    This checks if the file exists and if the file doesn't exist.
    """

    def pass_spf_record(self, mock_resolve):
        domain = "example.com"
        expected_record = 'v=spf1 include:_spf.example.com ~all'

        # Configuring the resolver
        mock_txt = MagicMock()
        mock_txt.to_text.return_value = expected_record
        mock_resolve.return_value = [mock_txt]

        result = get_spf_record(domain)
        self.assertEqual(result, expected_record)
        mock_resolve.assert_called_once_with(domain, 'TXT')

    def test_no_spf_record(self, mock_resolve):
        domain = "example.com"

        # Configuring the resolver
        mock_txt = MagicMock()
        mock_txt.to_text.return_value = 'not and spf record'
        mock_resolve.return_value = [mock_txt]

        result = get_spf_record(domain)
        self.assertEqual(result, second: 'No SPF record')
        mock_resolve.assert_called_once_with(domain, 'TXT')
```

Fig 8. Testing one of the reconnaissance tools.

6.3 Outcomes

The results of our comprehensive testing revealed that the cloud architecture and individual components performed as expected. The iterative testing of permissions and services in the cloud environment confirmed that the system was secure and functional. Adding components to the cloud proved to be straightforward with minimal configuration required. The process of uploading results to BigQuery was consistently successful, ensuring that data flowed through the pipeline. Metrics gathering was effective, with all recon tools successfully collecting and parsing data into CSV files. As a result of these tests, several improvements were identified and implemented, such as multi-threading the reconnaissance tools to speed up execution and spawning different instances of the same Cloud Run job to enhance the overall efficiency of the pipeline.

7.0 Version Control

We use GitHub as our centralized version control repository, allowing us to efficiently store and manage code, with all scripts, configuration files, and documentation maintained in organized repositories. GitHub Actions automate deployments, ensuring consistency and minimizing the risk of manual errors by deploying code directly to the Google Cloud environment.

As for our branching strategy, the main branch contains only thoroughly tested code to keep it stable. Developers create feature branches for new features or enhancements, so they can work independently. Automated testing and deployment are managed with GitHub Actions. Each pull request triggers automated tests to ensure new changes do not disrupt existing functionality. Once tests are successful, the code is automatically deployed to the Google Cloud environment, keeping the system stable and operational. Configuration files, including the Dockerfile for setting up the pipeline environment, are also version-controlled to ensure consistent deployments. We maintain comprehensive documentation and track changes diligently. A detailed changelog records significant updates, new features, and bug fixes, keeping developers and stakeholders informed about project progress.

8.0 Summary

The development of the Attack Surface Management System for Cybriant was a comprehensive effort to create a robust and scalable solution for profiling and analyzing the digital environments of small to mid-sized companies. The primary achievements include the successful integration of multiple security tools within a cloud-based infrastructure using Google Cloud services and the efficient processing and visualization of data through Looker Studio. The project met its objective of providing actionable insights to improve a company's security posture and reduce associated maintenance costs. The system was thoroughly tested to ensure reliability, efficiency, and the secure management of sensitive data.

8.1 Achievements

Key accomplishments include the successful implementation of a cloud-native architecture that efficiently manages and processes security data. The integration of Docker, Google Cloud Run, and BigQuery facilitated the seamless deployment and execution of various security tools. Custom Python scripts were developed to parse tool outputs, ensuring structured data storage in BigQuery. The system demonstrated efficient resource management, multi-threading capabilities for improved performance, and straightforward procedures for adding new tools. The automated pipeline effectively gathered, processed, and visualized security metrics, aiding companies in understanding their security posture and taking necessary remediation steps.

8.2 Lessons Learned

Throughout the development process, several valuable lessons emerged. First, the importance of cost management became evident, as high data ingestion and processing expenses could be significant if not carefully monitored. Implementing cost controls and optimization techniques for queries and job execution was crucial. Second, we learned that tool output inconsistencies could pose integration challenges, necessitating robust and adaptable parsing scripts. Third, ensuring proper permissions and security configurations in the cloud environment was critical to prevent unauthorized access and ensure the system's reliability. Finally, version control and continuous updates proved essential for maintaining compatibility with frequently updated tools, emphasizing the need for flexible and easily maintainable code.

8.3 Future Improvements

Looking ahead, there are several areas for future improvement. First, we plan to enhance cost optimization strategies by implementing more granular monitoring and resource allocation controls. This could include automated alerts for high-cost operations and more efficient query designs in BigQuery. Second, we aim to improve data processing speed by exploring additional multi-threading techniques and optimizing Cloud Run job execution. The integration of advanced analytics and machine learning models could provide deeper insights and predictive capabilities, further enhancing the system's value. We also intend to expand the suite of security tools and make the system more adaptable to emerging security threats. Finally, enhancing the user interface and visualization components will make it easier for security analysts and clients to interpret and act on the collected data.

9.0 Appendix

9.1 Project Plan

The pipeline will have two methods of invocation. First, a user can trigger the pipeline through interaction with a website by specifying a domain to be passed in as a parameter. Second, an automated process will iterate through a list of domains to maintain an up-to-date repository of information.

Both methods pass their parameters to the initial Google Cloud Service, which coordinates the invocation of subsequent jobs. The cloud service will invoke a Python script to update the Cloud Run Job container. Environment variables and the RUN command are updated based on which tools need to be run, determined on their last run time.

Once the Cloud Run Job container is updated, an iteration of that container is spawned. All jobs are invoked in parallel because of this. Each job iteration manages specific security tools and their respective metrics. One of the jobs to be invoked is the domain enumeration tool, Amass, which effectively identifies subdomains associated with a domain. Amass retrieves DNS records (MX, NS, A, and subdomain records). The results from the tool are parsed into a CSV using python.

The enumerated domains and subdomains will be evaluated for security misconfigurations. DNS servers will be checked for SPF, DKIM, DNSSEC, and TLS/SSL records, which indicate a basic level of security if found. Additionally, each domain will be checked for domain squatting attempts by malicious actors.

The A records will be utilized to provide a comprehensive view of the network associated with the domain. If no A records are found, an attempt will be made to retrieve the IP address directly. Open ports will be assessed for potential vulnerabilities in the services running on them. The full IP address range allocated to the domain will also be evaluated for any unintentionally exposed devices.

The website that is tied to the domain will be parsed for emails, and if any are found they will be checked for any potential credential breaches. The name server from the NS records will be queried if DNS records cannot be found through normal enumeration methods, and any AWS S3 buckets identified will also be analyzed.

The results from the various jobs will be stored in Google BigQuery, with tables created based on the data collected by the tools. These tables will be linked through a client identifier, enabling quick data aggregation. A ‘master table’ will be created that links together all of the other tables data. This will be done to facilitate easier integration with the secondary and tertiary objectives.

A secondary objective, if time permits, is to produce visualization aids. The parsed data from Chronicle will be used to create user-friendly visualizations, which will take on the appearance of a dashboard. This dashboard will have varying metrics and graphs to help understand the data that is found.

As a tertiary objective, the data found from the reconnaissance tools will be fetched from BigQuery and pre-processed before being sent to Google Chronicle (SIEM). Some client data will be parsed using Chronicle, with the results returned to BigQuery and added to the visualization dashboard.

9.1.1 Visual Project Outline

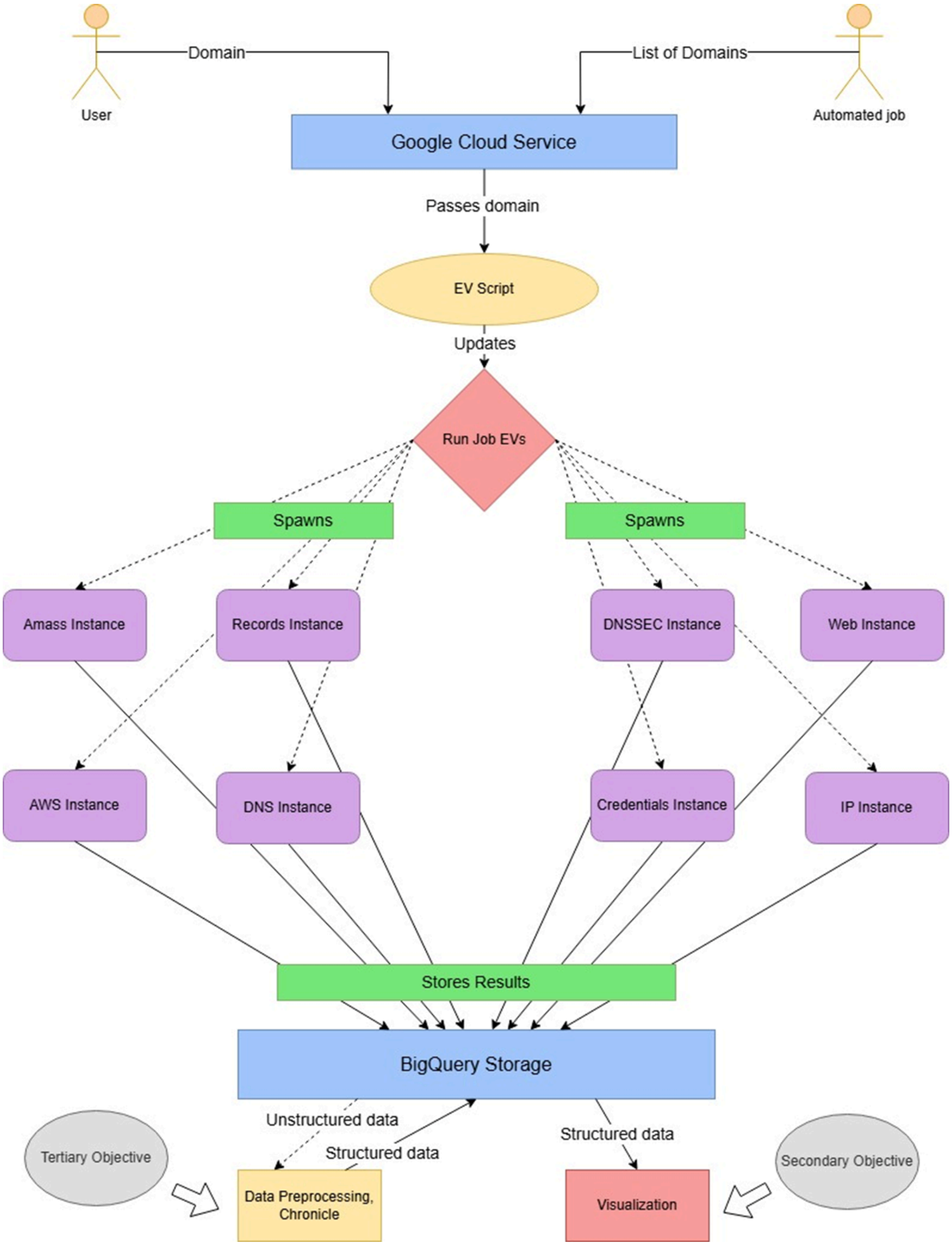


Fig 9. Visual representation of the project

9.1.2 Deliverables

Project Plan Work Breakdown: The project plan will outline the objectives and provide a timeline for achieving them. The project will be divided into modules, each with clear requirements.

Data Gathering Module: This module will involve various Python scripts invoked through Google Cloud Run jobs, utilizing various security tools grouped into similar categories.

Data Storage Module: Google BigQuery will store both raw and processed data. Client identifiers will be assigned to both data sets to facilitate filtering and organization.

Data visualization (Optional): As a secondary objective, the data that is found is to be inputted into a dashboard visualization. This visualization will feature various metrics from the data, logically grouped. Additional charts and graphs can be created to show changes over time.

Data Processing Module (Optional): As a tertiary objective, any raw data will be sanitized before being processed in Google's SIEM, Chronicle. Chronicle will parse a selective amount of the raw results from the data gathering module and convert it into structured results.

Extensibility Integration: Once the base project is established, additional targets will be introduced to gather more data sources, ensuring accurate averages across the data sets.

Software Testing Module: The various security tools will undergo unit testing to verify proper functionality, and automated integration testing will be implemented within the version control system.

Software Documentation: Detailed but concise comments will explain uncommon code patterns. User guides will be developed to help both developers and end-users navigate the software and understand its features.

Final Report Package: The final report package will include the complete software code, design documents, and technical specifications.

9.1.3 Milestone Events

1. **Project Planning** - By 09/06/24
2. **Architecture Design** - By 09/13/24
3. **First-Stage Development** - By 10/18/24
4. **Prototype Presentation** - By 10/24/24
5. **First-Stage Testing** - By 10/25/24

6. **Final Report Draft** - By 11/08/24
7. **Second-Stage Development** (**Data Integration**) - By 11/15/24
8. **Final Presentation** - By 11/15/24
9. **Second-Stage Testing** - By 11/22/24
10. **Final Report** - By 11/29/24

9.1.4 Meeting Schedule

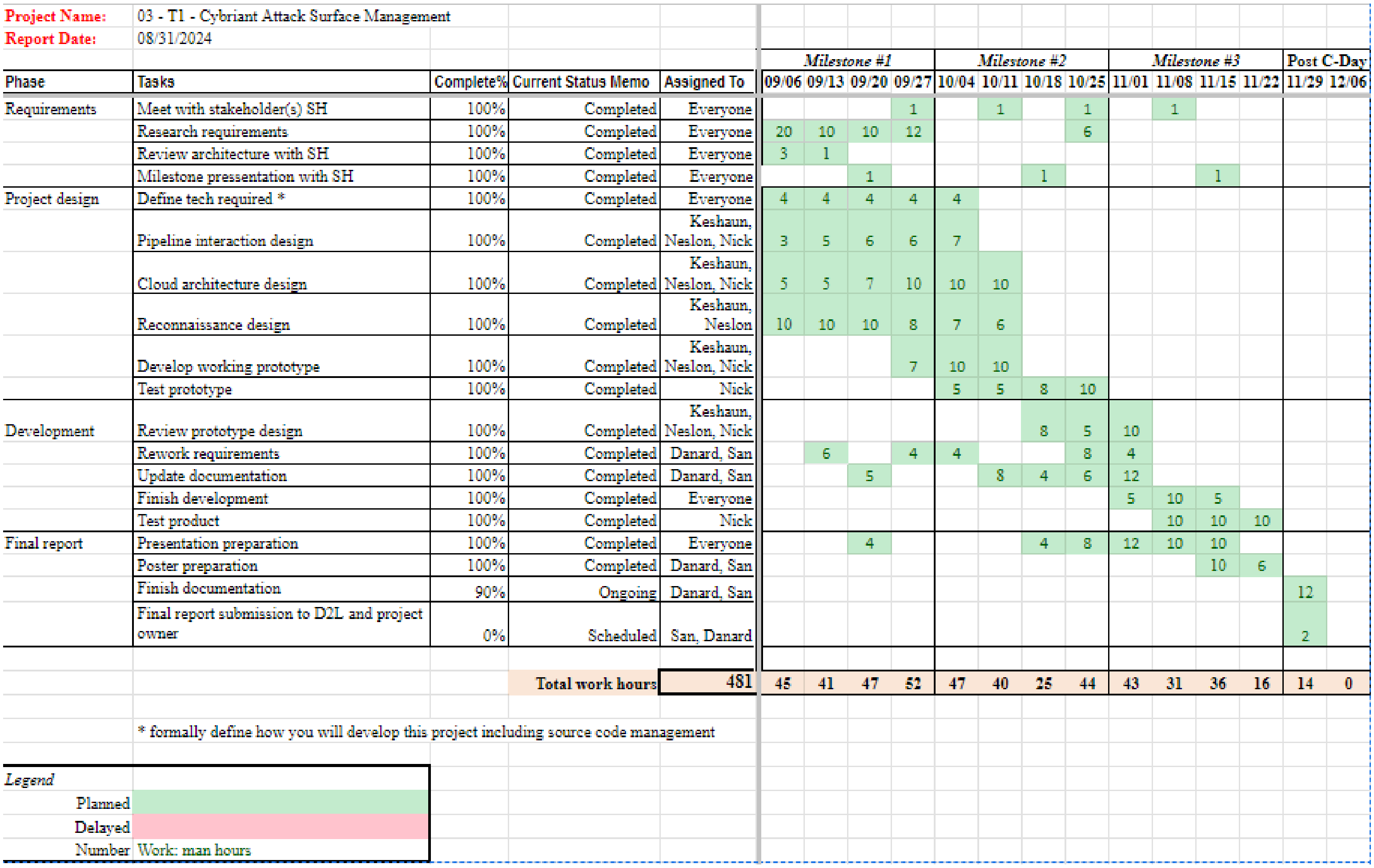
Status update meetings with Cybriant is bi-weekly on Fridays at 2:00 P.M. on Teams.

Milestone meetings are held every third Friday starting on the 20th of September at 2:00 P.M. on Teams to discuss the progression of the project.

Group meetings with Professor Perry is bi-weekly on Thursdays at 3:00 - 3:15 P.M. on Teams.

We have scheduled regular meetings with the Cybriant team and can request additional meetings as needed. Team communication will take place through a dedicated Teams channel, while both project teams can collaborate via the general channel. We will also meet with the professor to ensure the project stays on track.

9.2 Gantt Chart



Project Name:	03 - T1 - Cybriant Attack Surface Management																	
Report Date:	08/31/2024																	
					Milestone #1				Milestone #2				Milestone #3				Post C-Day	
Phase	Tasks	Complete%	Current Status Memo	Assigned To	09/06	09/13	09/20	09/27	10/04	10/11	10/18	10/25	11/01	11/08	11/15	11/22	11/29	12/06
Requirements	Meet with stakeholder(s) SH	100%	Completed	Everyone				1		1		1		1				
	Research requirements	100%	Completed	Everyone	20	10	10	12				6						
	Review architecture with SH	100%	Completed	Everyone	3	1												
	Milestone presentation with SH	100%	Completed	Everyone			1				1				1			
Project design	Define tech required *	100%	Completed	Everyone	4	4	4	4	4									
	Pipeline interaction design	100%	Completed	Keshaun, Neslon, Nick	3	5	6	6	7									
	Cloud architecture design	100%	Completed	Keshaun, Neslon, Nick	5	5	7	10	10	10								
	Reconnaissance design	100%	Completed	Keshaun, Neslon	10	10	10	8	7	6								
	Develop working prototype	100%	Completed	Keshaun, Neslon, Nick				7	10	10								
	Test prototype	100%	Completed	Nick					5	5	8	10						
Development	Review prototype design	100%	Completed	Keshaun, Neslon, Nick							8	5	10					
	Rework requirements	100%	Completed	Danard, San		6		4	4			8	4					
	Update documentation	100%	Completed	Danard, San			5			8	4	6	12					
	Finish development	100%	Completed	Everyone									5	10	5			
	Test product	100%	Completed	Nick										10	10	10		
Final report	Presentation preparation	100%	Completed	Everyone			4				4	8	12	10	10			
	Poster preparation	100%	Completed	Danard, San											10	6		
	Finish documentation	90%	Ongoing	Danard, San													12	
	Final report submission to D2L and project owner	0%	Scheduled	San, Danard													2	
			Total work hours	481	45	41	47	52	47	40	25	44	43	31	36	16	14	0

* formally define how you will develop this project including source code management

Legend	
Planned	🟩
Delayed	🟥
Number	Work: man hours

Fig 10. Completed Gantt Chart

9.3 Glossary

- **BitSight score** - A rating that corresponds to the amount of risk a system's security can mitigate.
- **Attack vector(s)** - Specific pathways an attacker can take to gain access to a system.
- **Profiling** - Analyzing system behaviors to gain an understanding of potential bad actors.
- **Exploit** - An exploit is a series of actions that take advantage of a vulnerability in a system. A person uses an exploit to gain access to otherwise inaccessible resources.

- **RESTful API** - This is an unofficial set of guidelines for passing data through a networked application. The Representational State Transfer suite consists of GET, POST, PUT, and DELETE operations.
- **POST** - One of the operations in the RESTful API guidelines. This operation is used to send data to a server.
- **GET** - Another one of the operations in the RESTful guidelines. This operation is used to fetch data from a server.
- **CLI** - The Command Line Interface is a system that enables a programmer to interact with the operating system. Commands are typed and executed in this system.
- **CVE** - The Common Vulnerabilities and Exposures list contains publicly disclosed cybersecurity vulnerabilities. Each CVE is assigned an identifier to facilitate information sharing.
- **CSV** - A Comma-Separated Value file uses commas to separate each value. This is a common format to store spreadsheet data.
- **BQ** - BigQuery is a serverless database offered in Google's Cloud Service. It has a speciality in dealing with large datasets and using real-time analysis.
- **AMASS** (**Asset Management and Security Scanner**) - An open-source tool for in-depth asset discovery, mapping the attack surface by finding subdomains, IPs, and other related information.
- **ASM** (**Attack Surface Management**) - A continuous security practice that identifies, monitors, and manages the potential attack vectors or surfaces of an organization's IT environment.
- **Cloud Compute**: Refers to cloud computing services provided by platforms like Google Cloud Compute, which offer scalable virtual machines and other resources for running applications.
- **Cloud Run** - A managed compute platform on Google Cloud that automatically scales containerized applications.
- **Google BigQuery** - A serverless, highly scalable, and cost-effective multi-cloud data warehouse designed for business agility, used for analyzing data using SQL.
- **Google Chronicle** - A cloud-based security analytics platform that provides threat intelligence, security telemetry, and data correlation to detect and respond to threats.
- **IP Address** - A unique address that identifies a device on the internet or a local network, used to direct data to its destination.

- **JSON** (**JavaScript Object Notation**) - A lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate.
- **NLTK** (**Natural Language Toolkit**) - A Python library used for processing human language data, often used in the ASM system for text analysis tasks related to threat intelligence.
- **Node.js** - A JavaScript runtime built on Chrome's V8 JavaScript engine, used for building scalable network applications.
- **OWASP** (**Open Web Application Security Project**) - An open-source security project focused on improving the security of software. OWASP ZAP is a widely used tool for web application security scanning.
- **Risk Assessment** - The process of identifying, analyzing, and evaluating the risk associated with discovered vulnerabilities, determining their potential impact, and prioritizing them for remediation.
- **Scanning** - The act of analyzing systems, networks, or applications for vulnerabilities, misconfigurations, or weaknesses that could be exploited.
- **Threat Intelligence** - Information about potential threats, including tactics, techniques, and procedures used by attackers, which is used to enhance security posture by identifying relevant threats.
- **Vulnerability** - A weakness or flaw in a system, application, or network that could be exploited by a threat actor to gain unauthorized access or cause harm.