**Research Prototype/MVP Implementation:** Decentralized and Graph-based Software Supply Chain Security (DG-SSCS)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**What is the project about:**

The project demonstrates “Decentralized and Graph-based Software Supply Chain Security” that provides various Supply chain security capabilities to enhance to detect the supply chain vulnerabilities across the project with developer sign into blockchain network. To achieve these capabilities, the application primarily depends on the GitHub GraphQL API to construct DAG style dependency graph and WEB3 to build a blockchain network among the developers.

**What are the features implemented:**

as below of a given GitHub project:

1. **Repository Dependency Graph in DAG model:**

To build a dependency graph in DAG style, the application fetches the dependencies of a user specified GitHub repository by using GitHub GraphQL API in the backend. The GitHub keeps verifying the repo dependencies and maintains dependency SBOM. The URL provided by GitHub is <https://api.github.com/graphql> along with application headers.

1. **SPDX Software Bill of Materials (SBOM):**

Software Package Data Exchange (SPDX) is an open standard for communicating software bill of materials (SBOM) information that supports accurate identification of software components, explicit mapping of relationships between components, and the association of security and licensing information with each component. The DG-SSCS application generates SPDX sbom using [**syft**](https://github.com/anchore/syft) tool from the repository and necessary fields are extracted from JSON object and integrated to Dashboard UI. SBOM automatically determines which package managers or build systems are being used by the software.

1. **Vulnerability Scanner:**

The application scans all the dependencies of a repo and determines the vulnerabilities of every package and provides the package ID, package name, CVE ID, Source URL, Severity and description of each vulnerability. To determine the vulnerabilities, the application uses a [**grype**](https://github.com/anchore/grype) package from GitHub. Grype supports input of **Syft**, **SPDX**, and **CycloneDX** SBOM formats. If Syft has generated any of these file types, they should have the appropriate information to work properly with Grype. It is also possible to use SBOMs generated by other tools with varying degrees of success. Two things that make Grype matching more successful are inclusion of CPE and Linux distribution information.

1. **Crypto function Detector:**

The objective of this module is to determent the all the Cryptographic functions that are utilized in the project and generate predefined JSON structure that includes line number of the Crypto function being used. To accomplish this task, this application used a [**crypto-detector**](https://github.com/Wind-River/crypto-detector) module developed by Wind River company. Detecting cryptography in the source code of open-source packages or libraries turns out to be a common problem for many of the software companies that include these packages in their products. This module also determines the Quantum-safe or not to every crypto function used in the project.

1. **Java Artifacts Multi-Graph (Directed):**

This module provides the multi-graph dependencies of Java projects dependencies for Java projects. Interdependencies between software modules are very difficult problems to solve. When your organization grows, these interdependencies multiply, potentially leading to low-cohesion and high-coupling among software modules. This is where [**Jarviz**](https://github.com/ExpediaGroup/jarviz) can help to understand these intricate dependencies down to the field level. Jarviz is a dependency analysis tool for Java applications. Since non-private methods in Java classes can be accessed by other classes, method calls can be intertwined creating manifold couplings among each other. Jarviz deeply analyzes Java bytecode to reveal these couplings in a user-friendly format. It scans binary artifacts using a custom classloader and generates a complete graph of dependency couplings between methods across multiple artifacts.

1. **Blockchain integration using WEB3 component:**

This module interacts with **Ethereum** protocol using **Web3.py,** a powerful API that can interact with blockchain.Ethereum is decentralized blockchain module that allows users to develop blockchain network using [**web3.py**](https://web3py.readthedocs.io/en/stable/quickstart.html)python library using fast API. The DG-SSCS application uses web3.py to sign the developers and their public repos using public key and private keys. Web3.py can help you read block data, sign and send transactions, deploy and interact with contracts, and other features as well.

1. **Container Imagen Scanner (SBOM and Vulnerabilities):**

DG-SSCS application can determine the SPDX SBOM and Vulnerabilities of container images. As of now the, the application supports only [Docker container images](https://hub.docker.com/search?q=&type=image). This module uses **syft** and **grpye** libraries to accomplish this task.

1. **Binary Image/file Analysis:**

This module analysis the Linux ELF or binary files and generates dependency graph between functions at assembly level language and determines the exceptions if any like buffer overflow or memory leakage based for different architecture support. [**angr**](https://github.com/angr/angr) is a multi-architecture binary analysis toolkit, with the capability to perform dynamic symbolic execution (like Mayhem, KLEE, etc.) and various static analyses on binaries. To generate a dependency graph, the agnr library create multiple in-built modules LIKE Control Flow Graph (CFG), data dependency graph (DDG), and value flow graph (VFG).

**What are the non-functional features?**

The DG-SSCS capable to export the analyzed data to .csv or .pdf formats. For example, once the SBOM or Vulnerabilities reports are extracted from the repo and be exported as .csv format. Similarly, the summary of Vulnerabilities per package can be exported from the Dashboard.

**What are bugs?**

The possible bugs can be addressed in two levels: External tools and application Bugs.

1. **External Tool Bugs:** The DG-SSCS, integrated various open-sourced tools and APIs.
   1. **GitHub GraphQL API**: 1. The number of dependencies or pull request or commits query beyond 100, the GraphQL throws an exception with rate limit. 2. If the git repo has, too many dependencies, the GraphQL unable to fetch the requested data and terminates with time limit error.

**Limitation**: The application limits with 100 dependencies only.

* 1. **Vulnerability module with Grpye tool**: The Grpye tool everyday checks/updates the Vulnerability database from the internet. If this takes too long to update with slow internet, the application raises an exception with connection error.

**Resolution**: Retry the application with GitHub repo url. Daily we need to check update the Vulnerability database using cron job.

* 1. **Web3.py:** 1. The Web3.py raises an exception, if the **Gas price** is limited at the Ethereum protocol. 2. Another possible error causes by web3: if the internet is slow, the unique hex code for the specific repo to add to blockchain will take longtime may causes a time out error or empty string inserted into database.

**Resolution:** Increase the **Gas prices** in the solidity configuration. Need a fast internet connection to create blockchain network.

* 1. **angr library:** It installed, tested and generated dependency graph only on Linux or Ubuntu platforms.
  2. **Binary Analysis Platform (bap):** Need to install all the dependencies on Linux or Ubuntu platforms. Then a bap will successfully be installed and generate a graph.

1. **DG-SSCS Applications Bugs/exceptions/limitations:**
   1. Rate limits with GraphQL API error
   2. User’s/developer need to detect new algorithms, need to add keywords manually to crypto-detector module
   3. Update syft and grpye tools very often at the user’s root directory.
   4. Currently, Java artifacts dependency graph is not dynamic. The graph is generated with pre-generated artifacts. To make dynamic, provide new \*.jsonl file python microservice.
   5. Binary file analysis, it supports only if the binaries are compiled on Linux or Ubuntu platforms.

**What are the test cases?**

As of now, there are no extremal test cases are written for the application. The test cases can be written using **pytest** library for backend. Currently, the microservices are written to verify the python REST APIs.

**Design**

Diagram, schematic

Description automatically generated

**Components**

1. Dependency graph (DAG style)
2. SPDX SBOM
3. Vulnerabilities
4. Local Repos from Database
5. Java Artifacts dependency multi-graph
6. Cryptography keys detector
7. Graph-BOM
8. Function-BOM
9. Decentralized Blockchain network
10. Container image scanner
11. Binary image/file analysis

**APIs/microservices**

1. /GetDependencyGraph
2. /GetSBOM
3. /uploadSign
4. /GetCryptoDetectors
5. /GetGraphHash
6. /GetRepoDetailsForBlockChain
7. /getBlockChain
8. /getAllBlockChainHash
9. /getGraphBOM
10. /getFunctionBOM
11. /GetContainerScanner
12. /getJavaArtifactGraph
13. /getBinaryImageScanner

**Algorithms**

1. [HMACs for Graphs (gHMAC)](https://ieeexplore.ieee.org/document/8118163).
2. Computed Quantum-safe for Crypto module.

**Any new ideas implemented?**

1. Implemented a Dependency graph in DAG style in python that is compatible with d3-graphs.
2. Implemented GraphQL API in python.
3. Implemented GraphBOM and FunctionBOM.
4. Implemented docker container scan.

**How to install?**

Create a **New GitHub Token** and it to **.env** file as ACCESS\_TOKEN in backend (python).

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

Graphical user interface, text, application, email

Description automatically generated

Currently the project is developed in 3-tier architecture. Frontend, backend, database. Each component has separate Dockerfile is made. The installation can be done in two ways:

1. **Dockerization:**

Step I: Identify Dockerfile in every component.

Step II: Build the docker-compose file to install and run the entire application with different ports that are being exposed.

1. **Individual installations and setup:**

* **Frontend app:** 
  + Download Nodejs and check npm installed or not.
  + Navigate to front-end folder.
  + `npm start` wiil through an exception with react-d3-
  + run the command `npm install -f` and then `npm start`. The application automatically opens the default browser with the port number. If the port is not available npm will wait until you change the port number.
* **Backend app:**
  + Install python3.9 or higher version.
  + Create a virtual environment using python. **Command:** `python3 -m venv envbackend`
  + Activate your own virtual python environment. **Command**: `source envbackend /bin/activate`
  + Install all the dependencies from requirennts.txt. **Command**: `pip3 install -r requirments.txt`
  + The flask environment by default run in debugging mode. The flask environments exported from .flaskenv file with ports, host, and app.
  + Run the flask python. **Command:** `python3 -m flask --debug run`

**NOTE**: backend-python: ACCESS\_TOKEN in **.env** file, update with a new git access token, if GitHub returns an error.

* **Backend Web3 app:**
  + Open another terminal or command line and follow the commands.
  + Create a virtual environment using python. **Command:** `python3 -m venv envweb3`
  + Activate your own virtual python environment. **Command**: `source envweb3/bin/activate`
  + Install all the dependencies from requirennts.txt. **Command**: `pip3 install -r requirments.txt`
  + Run the web3 client. **Command:** `python3 -m uvicorn client\_fastapiV3:app --reload --port 7000`
  + Do not change the port number for web3 client. Keep running it on 7000. If you would like to change, you need to add the port number in .env of backend module as well.
* **Backend binaryImageScanner app:**

**NOTE:** Use this service, when you are running the DG-SSCS app in MacOS only.

* + Open another terminal or command line and follow the commands.
  + Create a virtual environment using python. **Command:** `python3 -m venv binaryimage`
  + Activate your own virtual python environment. **Command**: `source binaryimage/bin/activate`
  + Install all the dependencies from requirennts.txt. **Command**: `pip3 install -r requirments.txt`
  + Run the web3 client. **Command:** `python3 -m flask run –host=” 10.200.102.145” --port 6000`
  + Do not change the port number, let the app run on 6000 only. If you would like to change, you need to update in **`GetBinaryImageScanner**` route URL in the backend module python **app.py**.

**How to invoke APIs**

**Using bash scripts:**

* Open terminal and navigate to microservices directory.
* Find the microservices written bash scripting with .sh extensions.
* Before you run the microservices, make sure the flask python is running with specific port number (default is 8000). If you change the port number, modify the same in every microservice.
* Run with /bin/bash or /bin/sh. **Command**: `bash file.sh`

**Command-line:**

* Using curl you can also invoke the APIs. Make sure the all the REST APIs are developed using POST methods only.
  + **Example:** To invoke a vulnerability scanner:

$ curl --noproxy '\*' -X POST http://localhost:8000/GetSBOM -H 'Content Type: application/json' -d '{"gitrepo":"https://github.com/cisco-open/flame", "sbom":"scan", "graphhash":""}'

**How to extend the prototype**

**Frontend Application**

1. Adding a new component at Menu bar

Graphical user interface

Description automatically generated

* 1. Open the <NavBar> component and add a new menu item.
  2. Create a new component that provides routing in the browser. Navigate the <App> component, import the component, add a new line in the <Route> component and place a component.

**Note**: Every component in must be surrounded by <ErrorBoundary> component.

* 1. All the Dashboard components are placed inside the “**Dashboard**” folder.

1. The DG-SSCS, app developed using multiple React components and version tested:
2. **Semantic-UI-React**: all the html and CSS components. (Version: **^2.1.3**)
3. **Ag-Grid-React**: to render the JSON object as table data (Version: **^28.2.1**).
4. **Rreact-d3-graph**: to render the Dependency graph in the dashboard (Version: **^2.6.0**).
5. **React-tostify**: to provide the notification (Version: **^9.0.8**).
6. **Axios**: to invoke the backend REST API with CORS (Version: **^1.1.2**).
7. UI to Backend connection
   1. Navigate and open the package.json,

à add a new line as “proxy”:”<backendServerIP>:<port>” before dependencies.

à Under **Scripts** section, “start-backend”:”cd <backend-root-folder>”

* 1. Make a REST API call to backend:

const Res = await axios.post("http://"+`${host}`+":"+`${port}`+'/GetCryptoDetectors,{

gitrepo: validateUrl(gitrepo),

}, 3000);

**Backend – flask python**

1. You can write own REST API service in app.py with flask python
2. First write the @app.route, that creates CORS from frontend to backend routing with request methods ‘GET’ or ‘POST’.
3. Once the route is defined, invoke a new definition that serves the frontend request and return the expected results.
4. **NOTE:** If you’re making a call from frontend to backend, the flask python keeps running the services infinitely. **Resolution:** invoke another definition and import the python modules in app.py.

const Res = await axios.post("http://"+`${host}`+":"+`${port}`+'/GetCryptoDetectors,{

gitrepo: validateUrl(gitrepo),

}, 3000);

**Where is it installed?**

* Make sure you install docker or docker desktop application need to run as background as a service.

1. Created a Dockerfile for every service (frontend, backend, web3, binaryimagescanner).
2. Created a docker-compose file to install to interact with application.

**VM Details**

IP Address: 10.200.102.145

username: jagdanda

password: ubusa@2021

path: /home/jagdanda/dg-sscs

1. All the modules are available in the **VM**.

**Concepts Learned**

1. Writing microservices in python.
2. Developed various bash scripts to consume the microservices.
3. Implemented **graph-based algorithm** with hashing.
4. Explored various open-source security tools.
5. Understand the workflow of software supply chain security (SSCS).
6. Explore and design the decentralized blockchain using Ethereum.
7. Learned the Common Vulnerabilities and Exposures (CVE) and its usage.
8. Quantified a Quantum-safe of all class of cryptographic algorithm.
9. Understand the integrating various tools and techniques to build a prototype.
10. Day-to-day work task review, identification of issues/bugs and its resolutions, and scheduling next tasks or enhancements in the module.
11. Design and develop a full-stack web app.
12. Step-by-step module integration and demonstration.

**Technologies Learned**:

1. Frontend technology - ReactJs
2. Backend technology - Flask Python
3. Decentralized Blockchain - Web3.py
4. DAG graph construction using react-d3-graph
5. Software bill of material using SPDX and CycloneDX
6. Understanding GitHub and it’s graphQL API
7. Docker and its associated tools
8. Deployment in the Linux remote server
9. Linux and MacOS commands
10. Ag-Grid react to render and export reports

**Issues Faced while Developing a Prototype:**

1. **Binary image scanner:**
   1. The `angr` library supports currently tested and supported only from Linux or Ubuntu platform, however it the binary is compiled on Mac OS, the module fails to generate a dependency graph.
2. **Vulnerability Scanner:**
   1. This prototype currently uses `grype` a vulnerability scanner of a local folder or a docker container. The tool pulls the latest database from the server and updates the database.

**Resolution**: A **cron job** can be crated to update the database every **at 8am** with below command to cron job: 0 8 \* \* \* /path/to/grype db update