Project Title: IoT Security Posture Web App

Team Name: IoT-VAS

Team Members:

Jack Cartwright
- Backend scanning automation/API work

Karl Thimm
- Front end UI development and data presentation/API work

Project Summary:

For our senior design project we decided to create an Internet of Things Vulnerability Scanner. This can be done using open-source software like OpenVAS and greenbone python-gvm API. OpenVAS is an open-source vulnerability scanner that is used for scanning machines based on their ip addresses. OpenVAS then returns a vulnerability report containing all of the weaknesses on the system you scanned, whether it be a computer or another device on the network. We wanted to take OpenVAS's components and then integrate them into our own scanner. Using the Greenbone API we would call the scanner and perform its operations from our own web app. We would need to create everything for our web environment from backend to frontend, the frontend being our web app to call the scanner functions.

To host our backend we decided to use docker because of how easily we could have the containers communicate with each other. We needed flask, nginx and the OpenVAS to all work together to be able to perform the operations we needed. Flask is needed to host our web page templates (html files) and nginx was needed to store our javascript files which were used to call upon the API to run the scans.

Our plan for our web app was very similar to how the actual OpenVAS web application works. The user would first need to create a target which is done by naming the target and then specifying an ip address for the device you want scanned. The user then has to create the task where they select the scanner type (OpenVAS) and the configuration of the scanner (full and fast, base, etc…) Once the task was created the user can start the task which will start the scan on the target. The user can then check the progress of the scan using a progress report button on the web app. Once the scan was complete the user could request the report from the scanner which would open a new tab in the web browser and display the PDF containing all the results being the vulnerabilities found on the device.

Problem Statement: Currently there is no such thing as an Internet of Things vulnerability scanner. Internet of Things devices can be very vulnerable devices despite having one or a few functionalities. These vulnerabilities can come from things like weak authentication, outdated software/firmware and insecure communication across networks. These vulnerabilities range in a scale of 0-10 severity. A low severity means an experienced hacker can access some files in a system but nothing serious. A 10 severity means that an experienced hacker would be able to gain access to the entire system with administrator permissions. Our customers would be anyone on the web who has a use for a vulnerability scanner for IoT devices. We could even add more scanners, not just OpenVAS and let customers access the application on a subscription basis.

Goals, Objectives and Constraints:

Our goals and objectives were to develop an app to perform automated scans/network discovery on IoT devices. The web application needs to be able to scan for devices and discover IPs. Then scan selected IPs for vulnerabilities using OpenVAS and put the vulnerability history on a graph, compare graphs from scan to scan to see if vulnerabilities are getting better or worse over time. Also it needs to provide solutions for common vulnerabilities if available. A success would be a proof of concept web interface capable of scanning selected IPs for vulnerabilities and tracking how many and how severe the vulnerabilities are over time as well as providing solutions where available. A failure would be having the bare bones of the project working, but we are not able to create targets, run scans and view the results. By the end of fall we needed the project to be a very early prototype but be able to function and run scans even if it is a very manual process to get scans started.

In order to make this project come to life there were a lot of design constraints we had to consider and work our way around. Our first constraint when planning we had to consider was our budget. We were only allowed to spend a few hundred dollars, but lucky for us we were using all open-source software which means everything is free. The second constraint we had to consider was the limited time we have to complete this project. We needed to plan out the steps we were going to take each week and figure out where we wanted to be by the end of the fall. Everything that we couldn't get done in the fall would need to be done in the spring. Two semesters is not a lot of time for a large project like this so we needed to design the functionalities we would want to not be too hard but also not too easy. A third constraint we needed to discuss was having a minimal user interface so the app would be easy to use. If there was too much on the web page and everything was all over the place it would be very confusing for a user to figure out how to use the web application. We needed to design a web page that was easy to navigate and would be very easy for a user to follow. We decided to accomplish this by using different html files for each web page, we would have a login page which would then bring you to the scanning page where all the steps to start a scan would be in the order you needed to fill everything out. One other constraint we needed to figure out was if we would be able to add new scanners. Out of the gate OpenVAS comes with a CVE scanner and the OpenVAS scanner but we had to figure out if we could add more. We decided to use docker to host our application which would allow us to easily add the OpenVAS scanner to start and would allow us to add many more once we had the first one working. The last constraint we needed to discuss was how much server space we would need for our database. We decided that we would use SQL for our database to just store user accounts and their information which would not need a lot of storage.

In the spring, we focused on enhancing the application's functionality. We implemented automated network discovery using nmap and a Python library called subprocess to execute terminal commands. This feature parses the output, displays device IPs and names, and creates targets for scanning. We also integrated Postgres databases to manage user logins, targets, tasks (scans), and report PDFs, allowing us to add entries and display information seamlessly within the web application. To improve usability, we created dedicated pages for tasks, targets, and reports instead of consolidating everything on a single page. This modular approach enhanced navigation and user experience.

The system requirements included running on open-source software to stay within budget, performing automated network discovery and vulnerability scans, maintaining an intuitive and easy-to-navigate interface, supporting the addition of new scanners, and scalable database management. Security measures were essential to protect user data and scanned information. We assumed that users would have a basic understanding of IoT devices and network security, that the application would run in an environment with sufficient network access to perform scans, that OpenVAS and other integrated tools would remain open-source and free to use, and that Docker would be used to manage the deployment and scalability of the application.

Fall/Spring Plan Schedule

Week of	Week #	Activity	Milestone
9/18/2023	4	Investigate existing TCP API for OpenVAS	
9/25/2023	5	Discuss UI/Frontend Design	

10/2/2023	6	Perform scans using API and automation	Scanning API functionality
10/9/2023	7	Store scan results in database	
10/16/2023	8	Allow loading database results into UI	
10/23/2023	9	Allow starting scans from UI	Full basic frontend functionality
10/30/2023	10	Add scheduling to the scans to the backend	
11/6/2023	11	Create the UI for scheduling scans	
11/13/2023	12	Prototype has Full Functionality	Full Functionality
11/20/2023	13	THANKSGIVING	
11/27/2023	14	Prep for Final Report/Presentation	
12/4/2023	15	Prep for Final Report/Presentation	
12/11/2023	16	Final Report/Presentation	Presentation
WINTER BREAK		WINTER BREAK	
2/5/2024	1	Explore authentication mechanisms and libraries	
2/12/2024	2	Add credentials	Multi User
2/19/2024	3	Determine a suitable method for network discovery	
2/26/2024	4	Get Network Discovery Working (Scan IPs on a network)	Network Discovery
3/4/2024	5		
3/11/2024	6	Add more scanners	More comprehensive results
3/18/2024	7		
3/25/2024	8	SPRING BREAK	
4/1/2024	9	Combine results of multiple scanners into one comprehensive overview	
4/8/2024	10		
4/15/2024	11	Deploy the app in a production environment	Fully deployed finished product

4/22/2024	12		
4/29/2024	13	Prep for Final Report/Presentation	
5/6/2024	14	Prep for Final Report/Presentation	
5/13/2024	15	Final Report/Presentation	Presentation

Approach:

When planning the design for our project we had a lot of things we needed to consider. First we needed to decide what software we would use for our scanner and how we were going to host our web app. We decided to use the open-source software OpenVAS and we originally planned on using firebase for hosting. Firebase is a platform that allows for hosting web applications with ease. It allows for easy integration of authentication which would allow our users to have their own accounts and also allows you to implement a database which we planned on using to store different users and scan information specific to each user. This means that “user 1” could login to the web application and see only their scans and not other users such as “user 2”. This is essential to have in our IoT scanner because if users can see every single scan run by any other person then they would be able to see the vulnerabilities in thor devices, exposing confidential information.

Our next step was how we were going to host our files needed for the software to function. We decided to use docker containers which would allow each container to communicate with each other easily. Docker also would allow us to easily deploy the software on any system and would allow us to roll back to earlier versions if we ran into configuration issues or broke everything when adding new functionalities. We then needed to figure out how we would interact with the scanner from our web application. After looking through a lot of different open-source software we decided to use the Greenboen python-gvm API. This would allow us to make calls from the web app to our scanner in order to function as we wanted it to. Once we had the API integrated into docker and we could run curl commands from the terminal such as calling the scanner version with an API call we needed to have our web app communicate with the scanner. Since firebase uses HTTPS and our docker was running on a localhost it was near impossible to get these two to work interchangeably. After long discussions we decided that firebase was not the best option to use and we instead moved towards using Flask to host our html files instead using jinja templates. We would be able to run flask using a docker container so this would allow our backend and frontend to communicate. Once switched to strictly docker containers we hosted our webpage on the localhost website on port 8888 (http://localhost:8888). Once we had this up and running we created an account to be able to login using flask login and then began to test our API functionality on the browser.

To test the API we first made an API call to get the scanner version which would print the current version in the console. Once this was working we moved to adding API calls to get the scanners and scanner configurations which are all “GET” requests. We accomplished all of this by writing javascript functions to call to the API and print the results, which was run once a button on the webpage was clicked. Then we had to move to adding the “POST” API calls to create a target, create the task, run the scan, print the progress and get the results of the scan. These are a little different then the scanner “GET” functions because we needed to send information to the scanner instead of just pulling information. Working through one function at a time we were able to get functionality with all of the components needed to complete the scanning process.

In the spring, we expanded our application's capabilities. We added network discovery using nmap and a Python library called subprocess to run terminal commands. This feature parses the output, displays device IPs and names, and creates targets for scanning. Additionally, we integrated Postgres databases to manage user logins, targets, tasks (scans), and report PDFs. This structure allowed us to add and display information seamlessly within the web application. To enhance usability, we designed dedicated pages for tasks, targets, and reports, rather than consolidating everything on a single page. This modular approach improved navigation and user experience. Building on our initial setup, we implemented network discovery by integrating nmap into our Docker containers. Using Python's subprocess library, we automated terminal commands to identify device IPs and names, creating new scan targets dynamically. This enhancement allowed for real-time network mapping and target creation directly from the web interface.

We further enhanced the backend by integrating Postgres databases. These databases stored user logins, scan targets, tasks, and report PDFs. This structure not only supported secure user authentication and data management but also streamlined the process of adding and displaying information within the web app. The integration of Postgres required writing and executing SQL scripts, ensuring robust and scalable data handling. To ensure a clean and user-friendly interface, we redesigned the web application to feature dedicated pages for tasks, targets, and reports. This modular approach improved user navigation and clarity, making it easier for users to manage their scans and view results. Each page was developed using Flask and Jinja templates, ensuring consistency and ease of use across the application.

Testing and debugging were critical throughout the project. For the API integration, we started by testing "GET" requests to ensure the scanner configurations were retrieved correctly. JavaScript functions facilitated these API calls, with results printed to the console for verification. Once the "GET" requests were functional, we moved on to "POST" requests, which involved more complex interactions. Each function was tested individually, with extensive logging to track data flow and identify issues.

To ensure our web application was functioning correctly and to diagnose issues, we used Adminer. Adminer is a lightweight, easy-to-use tool for managing databases, offering a simple web interface to perform database operations. By using Adminer, we were able to directly inspect the data stored in our Postgres database, which was crucial for verifying the integrity and accuracy of our application's data handling processes.Adminer allowed us to verify whether the data submitted through our web app's forms was being correctly added to the Postgres database. We could check if user login information, scan targets, tasks, and report PDFs were properly stored in the relevant tables. This step was essential to confirm that our "POST" API requests were functioning as expected and that the data being sent from the web app was accurately inserted into the database.Additionally, Adminer helped us diagnose issues when there were discrepancies between the data stored in the database and what was displayed on the web app. For example, if a user reported that their scan results were not showing up, we could use Adminer to check if the data was present in the database. This process helped us determine whether the issue was with the data insertion (i.e., the web app not properly adding data to the database) or with the data retrieval (i.e., the web app incorrectly pulling information from the database).

By providing a clear and direct view of the database contents, Adminer was instrumental in our troubleshooting process. We could quickly identify and resolve issues related to data integrity and display, ensuring the application's reliability and performance. This tool also facilitated the testing of SQL queries and database schema changes, further supporting the development and maintenance of a robust backend for our application. Overall, Adminer played a vital role in our development workflow, aiding in both debugging and verifying the accuracy of our database operations.

Challenges:

In order to complete this project in our vision there are many challenges that need to be overcome. Coming into this project both Jack and I had little experience in designing web interfaces, integrating APIs, configuring a backend and getting them all to work together. The first step to overcoming this is to do research and understand how all of these components function. After researching we understood how each piece of this project was going to fitr together, our backend would be our server, database and API/Javascript functions. The frontend would be what the user sees on their end which is how the scans would be created. FInally the API is what would “glue” the frontend and backend together, allowing them to communicate back and forth.

Understanding was only part of the problem, the next problem was how we were going to actually accomplish this. Overcoming this problem was a slow learning process. Doing a lot of research we decided that using docker would be the easiest way to host our frontend and backend to allow them to communicate. Once we had docker up and running with our flask, nginx and OpenVAs containers we then had to figure out where to start. We had no idea how to get our frontend to interact with our backend. The best way to learn how to do this would be to try and fail over and over again until we made progress. We integrated little by little, first making a basic webpage that we could add some functionality to. The next step was learning how we can request or send information to the backend using the API. After doing our research we found that we could use javascript functions and receive information from the API. First starting simple by adding buttons that would call the javascript function upon clicking it and getting that to function before moving on to the more difficult functions we would need to create for our webpage to have functionality.

In the spring, we tackled new challenges by expanding our application's capabilities. One significant addition was network discovery, which automatically uses nmap and a Python library called subprocess to run terminal commands, parse the information, and display device IPs and names, creating targets for scanning. Integrating this feature was challenging because it required ensuring that the subprocess commands executed correctly and securely within our Docker environment. Parsing the output accurately and displaying it on the web app also required careful handling to avoid errors and ensure that the data was presented clearly to the users. Another major addition was integrating Postgres databases to manage user logins, targets, tasks (scans), and report PDFs. This required us to set up a robust database schema and ensure that our web app could interact with the database seamlessly. We had to write SQL scripts to handle data insertion, retrieval, and updates efficiently. One challenge we faced was ensuring data consistency and integrity, especially when multiple users interacted with the application simultaneously. We had to implement proper error handling and validation checks to ensure that the data in our database remained accurate and reliable.

Additionally, we restructured our web application to create dedicated pages for tasks, targets, and reports instead of consolidating everything on one page. This modular approach improved user navigation and clarity but also presented challenges in maintaining a consistent user experience across different pages. We had to ensure that the navigation between pages was smooth and that the data displayed was always up-to-date. This involved a lot of testing and debugging to catch any inconsistencies or errors in the data flow between the frontend and backend. Overall, integrating these new features in the spring was a complex process that required careful planning, extensive testing, and a lot of patience. By tackling these challenges step-by-step and learning from our mistakes, we were able to significantly enhance our application's functionality and provide a more robust and user-friendly experience.

Design:

The overall design of our project includes a backend and a frontend. The backend is for our database, server, javascript functions and our API and our frontend is what the user will see and interact with in order to run scans. Docker is the application that ties this all together, allowing for them to communicate and interact. Docker containers are lightweight, portable, and self-sufficient execution environments used to package and run software applications along with their dependencies. Docker allows for easy development of our software, if we run into configuration issues we can easily roll it back and it allows multiple containers to run at the same time and communicate. Our backend is hosted by the nginx container which is a web server for serving static files and can also be used as a reverse proxy. Nginx serves things like Javascript and CSS which gives our web page functionality and makes it look nice. CSS gives the webpage things like color, fonts, text size and can put boxes around our components. CSS is very important for having a nice user interface that is easy to navigate and use.

Our frontend which is hosted by flask on a localhost website allows users to use the application. Flask is a web framework for python commonly used to build web applications. It serves as the backend or server-side component of our application. Handles various functionalities and interactions between greenbone API (python-gvm) and our web application Allows specific requests to be made from the web app, performs the necessary operations or calls to the API to fetch or manipulate data and returns the results in either JSON or PDF format. Even though flaks is a backend, it allows us to render our html files so we also used it as a frontend for simplicity. Hosting the html files through nginx complicates the code and makes the process of creating the web application much harder.

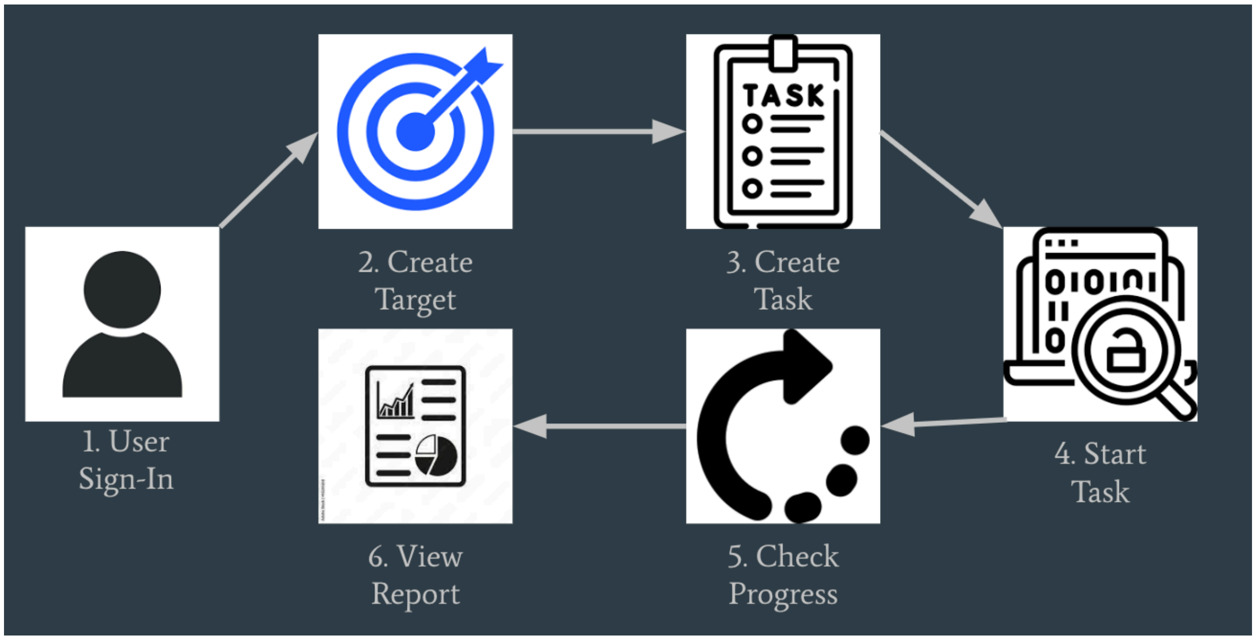
Our frontend or user interface works by first allowing a user to sign in. The user then is brought to a webpage with many different functionalities. The first thing on the webpage is “create a target” which is where the user enters a name of the machine and the ip address of said machine, the name does not matter. The user can then use this target to create a task which is where a name for the task is entered (name does not matter), and the scanner type and configuration is selected. The scanner type and configuration have hard coded IDs that are passed to the scanner using the API. To hardcode these ids into our webpage we created a dropdown menu with names that display, when a name is selected the id that is attached to that name is passed to OpenVAS which allows it to know which scanner and which configuration to run the scan with. Once the task is created the user can then choose to start the task which starts the scan on the target machine. The user can check the progress of the scan while it is running, this works by the task ID being passed to the scanner using the API which then returns the progress attached to that task ID. Once the scan is complete which can take as long as 4-5 hours the user can then request the results. The user calls for the report by passing the report ID into the API which then gets the PDF from the scanner and displays it to the user in a new tab on their web browser. Everytime the user interacts with the webpage the first thing called is our Javascript functions which are used to interact with the API. The API then interacts with the scanner and either gives it information, grabs information or does both at the same time. The information that needs to be returned is then grabbed by the Javascript in order to be displayed on the webpage, whether that be the reports of scans or the ids of specific targets and tasks.

In the spring, we tackled several new challenges and added significant features to our application. One of the major additions was network discovery. We integrated nmap, a powerful network scanning tool, and used a Python library called subprocess to run terminal commands. This feature automatically scans the network, parses the information, and displays the device IPs and names, creating targets for further scanning. Implementing this feature was challenging because it required us to ensure the subprocess commands executed correctly within our Docker environment. Parsing the nmap output accurately and displaying it on the web app also required careful handling to avoid errors and ensure clear presentation to users. Another major enhancement was integrating Postgres databases to manage user logins, targets, tasks (scans), and report PDFs. This addition required us to set up a robust database schema and ensure seamless interaction between the web app and the database. We wrote and executed SQL scripts to handle data insertion, retrieval, and updates efficiently. Ensuring data consistency and integrity, especially when multiple users interacted with the application simultaneously, was a significant challenge. We implemented proper error handling and validation checks to maintain accurate and reliable data. To improve the user interface and overall user experience, we redesigned our web application to create dedicated pages for tasks, targets, and reports instead of consolidating everything on one page. This modular approach made navigation more intuitive and allowed users to manage scans and view results more easily. Each page was developed using Flask and Jinja templates, ensuring consistency and ease of use across the application. Throughout the spring, testing and debugging were critical to ensure the reliability and functionality of the new features. We used Adminer, a lightweight tool for managing databases, to inspect data directly and verify that requests from the web app were correctly added to the Postgres database. Adminer helped us identify whether issues were due to the web app not properly adding data to the database tables or if the web app was incorrectly pulling information from the tables. This clear view of the data was instrumental in troubleshooting and maintaining the application's reliability.

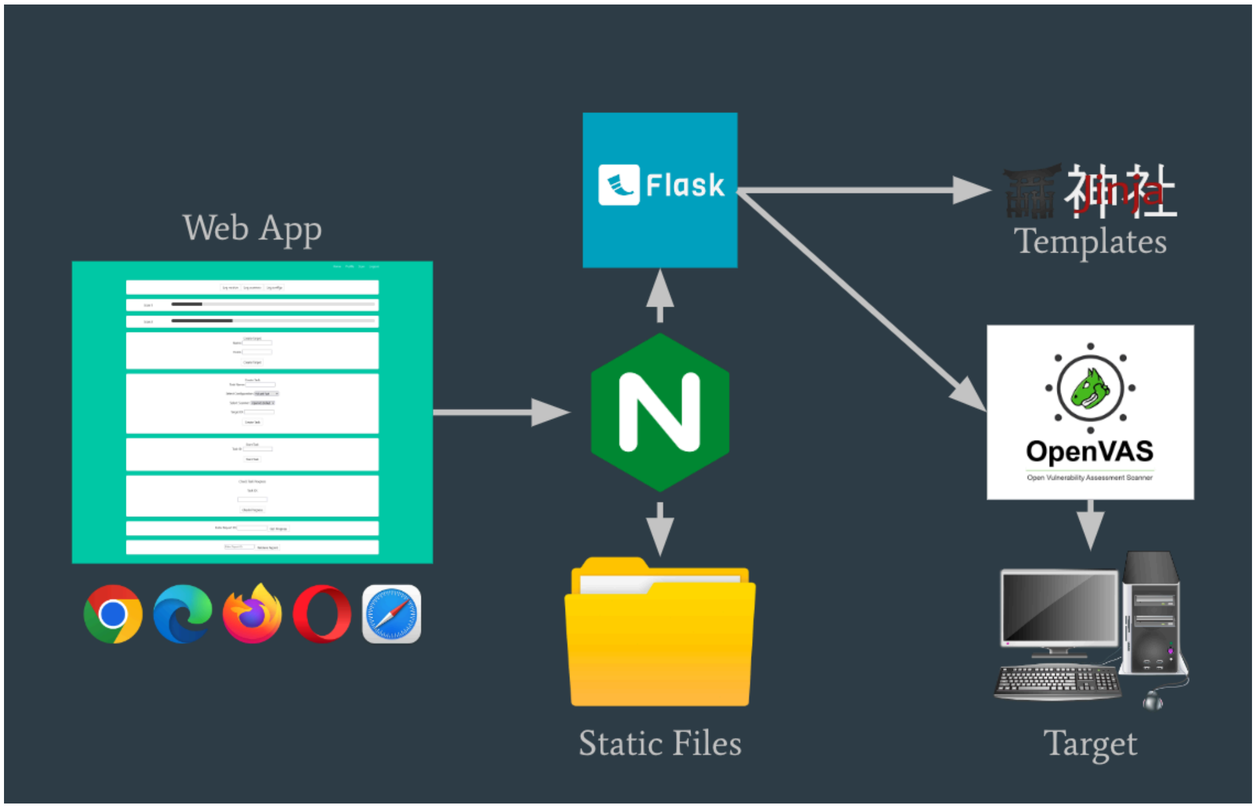
In addition to these developments, we made significant adjustments to the python-gvm API to better suit our web application’s needs. While the python-gvm API provided a robust interface for interacting with OpenVAS, we needed it to seamlessly integrate with our Postgres databases. To achieve this, we included SQL commands within the API functions to handle data operations directly. This allowed us to push information to the database when new scans were initiated and pull information from the database to display results and updates to the users. Integrating SQL commands into the API streamlined our data management process, reducing the need for separate database handling code and ensuring that all interactions with the scanner and the database were tightly coupled. For example, when a new scan target was created, the API not only communicated with OpenVAS to set up the target but also executed SQL commands to store the target details in the Postgres database. Similarly, when retrieving scan results, the API pulled the relevant data from the database, ensuring that users had up-to-date information available on the web application. This approach minimized data redundancy and improved the efficiency of our application. By embedding SQL commands within the API, we maintained a consistent flow of data between the scanner, the database, and the frontend, enhancing the overall reliability and performance of our system.

By the end of the spring, these enhancements had significantly improved our project's functionality and user experience. The network discovery feature allowed users to easily identify and target devices on their network for scanning. The integration of Postgres databases provided a robust and scalable solution for managing user data and scan results. The redesigned user interface made it easier for users to navigate the application and manage their scans effectively. Overall, these improvements brought us closer to realizing our vision for a comprehensive and user-friendly web application for IoT vulnerability scanning.

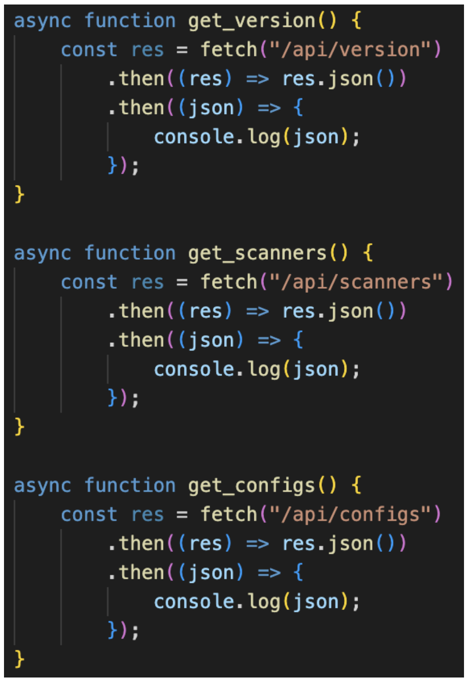
Fall Pictures



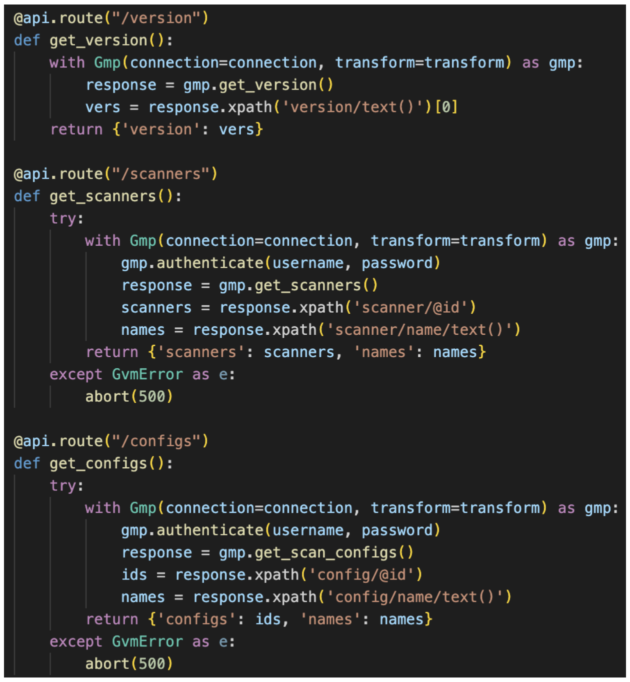
User Interface Functionality



Backend Functionality



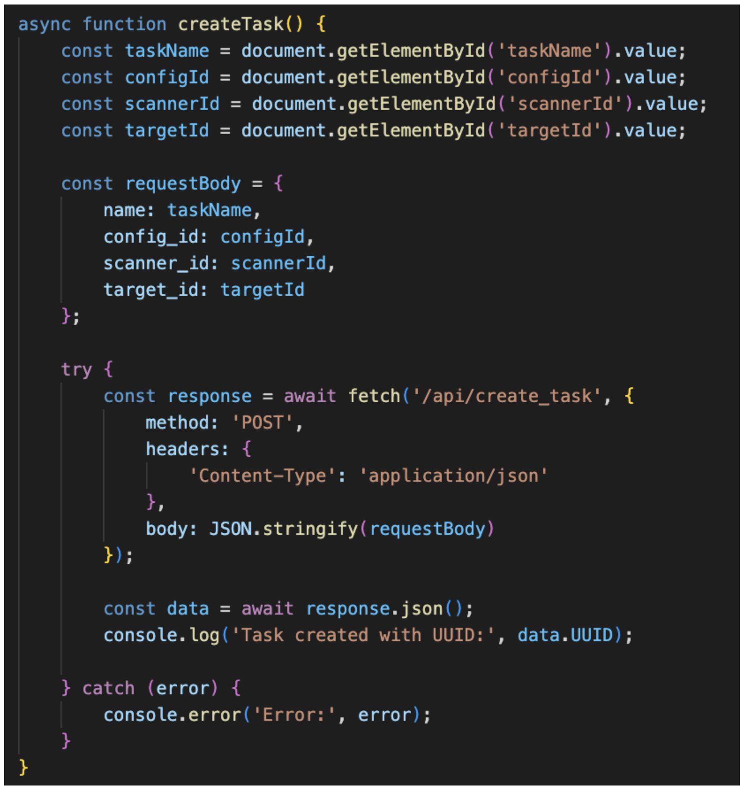
```javascript
async function get_version() {
    const res = fetch("/api/version")
        .then((res) => res.json())
        .then((json) => {
            console.log(json);
        });
}

async function get_scanners() {
    const res = fetch("/api/scanners")
        .then((res) => res.json())
        .then((json) => {
            console.log(json);
        });
}

async function get_configs() {
    const res = fetch("/api/configs")
        .then((res) => res.json())
        .then((json) => {
            console.log(json);
        });
}
```



```python
@api.route("/version")
def get_version():
    with Gmp(connection=connection, transform=transform) as gmp:
        response = gmp.get_version()
        vers = response.xpath('version/text()')[0]
    return {'version': vers}

@api.route("/scanners")
def get_scanners():
    try:
        with Gmp(connection=connection, transform=transform) as gmp:
            gmp.authenticate(username, password)
            response = gmp.get_scanners()
            scanners = response.xpath('scanner/@id')
            names = response.xpath('scanner/name/text()')
        return {'scanners': scanners, 'names': names}
    except GvmError as e:
        abort(500)

@api.route("/configs")
def get_configs():
    try:
        with Gmp(connection=connection, transform=transform) as gmp:
            gmp.authenticate(username, password)
            response = gmp.get_scan_configs()
            ids = response.xpath('config/@id')
            names = response.xpath('config/name/text()')
        return {'configs': ids, 'names': names}
    except GvmError as e:
        abort(500)
```

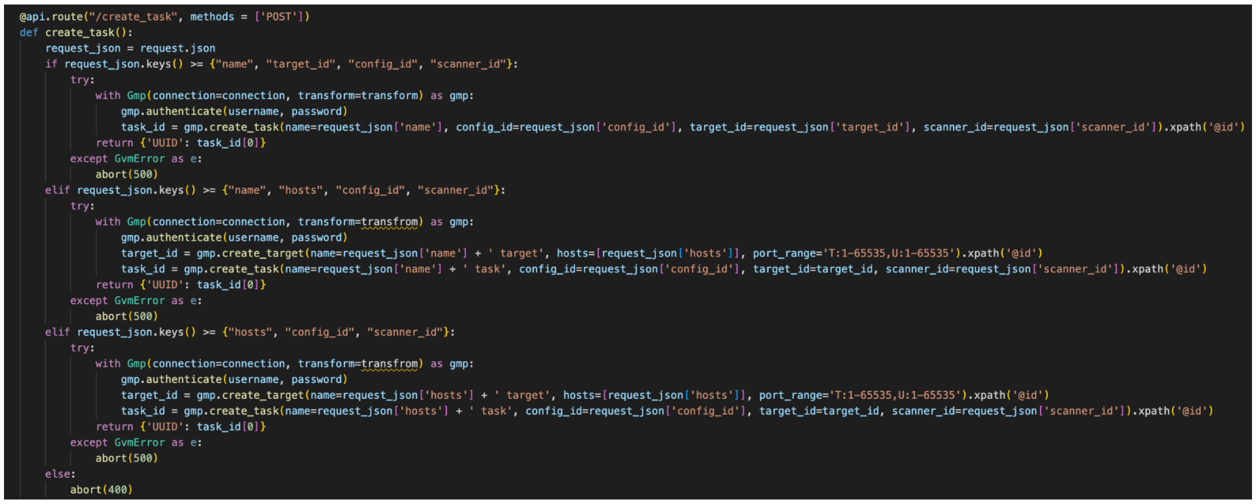
Javascript for Web App **Python API Functions**

-These are the functions we tested early such as the get version call.



```javascript
async function createTask() {
    const taskName = document.getElementById('taskName').value;
    const configId = document.getElementById('configId').value;
    const scannerId = document.getElementById('scannerId').value;
    const targetId = document.getElementById('targetId').value;

    const requestBody = {
        name: taskName,
        config_id: configId,
        scanner_id: scannerId,
        target_id: targetId
    };

    try {
        const response = await fetch('/api/create_task', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(requestBody)
        });

        const data = await response.json();
        console.log('Task created with UUID:', data.UUID);

    } catch (error) {
        console.error('Error:', error);
    }
}
```

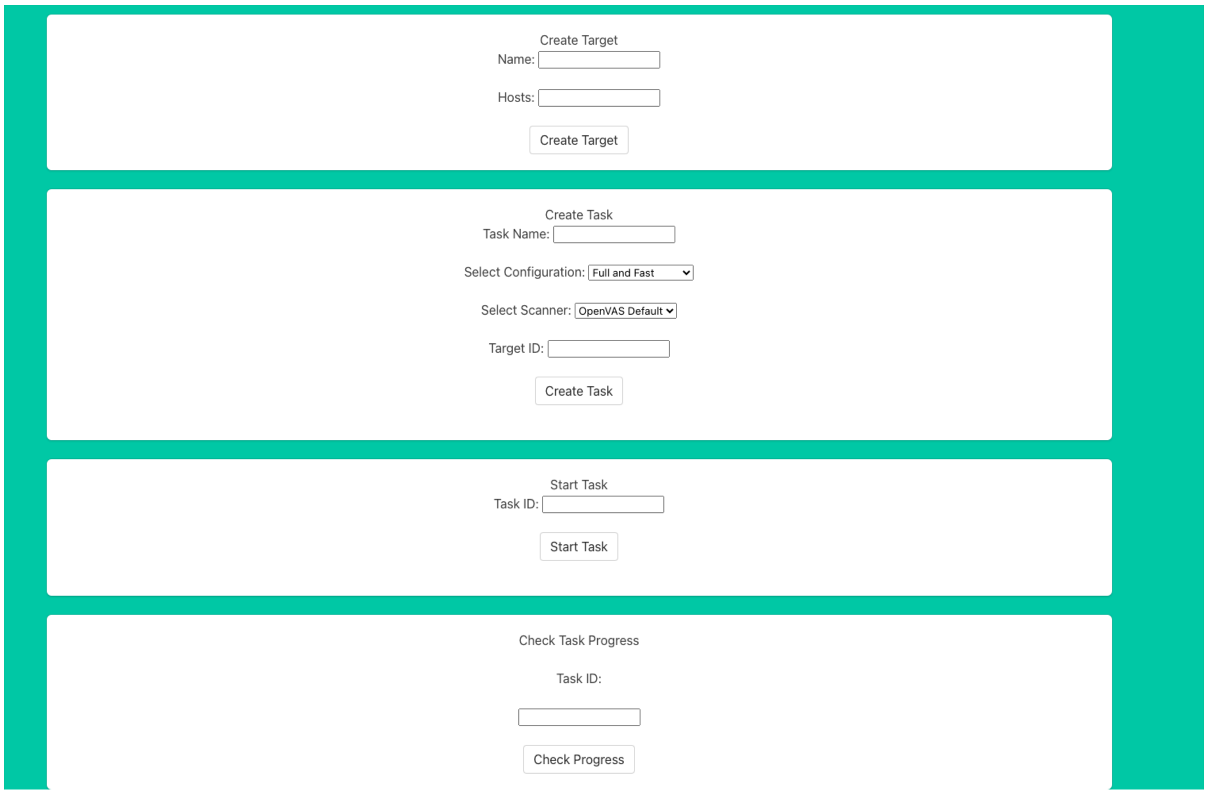
Javascript Create Task Function



```python
@api.route("/create_task", methods = ['POST'])
def create_task():
    request_json = request.json
    if request_json.keys() >= {"name", "target_id", "config_id", "scanner_id"}:
        try:
            with Gmp(connection=connection, transform=transform) as gmp:
                gmp.authenticate(username, password)
                task_id = gmp.create_task(name=request_json['name'], config_id=request_json['config_id'], target_id=request_json['target_id'], scanner_id=request_json['scanner_id']).xpath('@id')
            return {'UUID': task_id[0]}
        except GvmError as e:
            abort(500)
    elif request_json.keys() >= {"name", "hosts", "config_id", "scanner_id"}:
        try:
            with Gmp(connection=connection, transform=transfrom) as gmp:
                gmp.authenticate(username, password)
                target_id = gmp.create_target(name=request_json['name'] + ' target', hosts=[request_json['hosts']], port_range='T:1-65535,U:1-65535').xpath('@id')
                task_id = gmp.create_task(name=request_json['name'] + ' task', config_id=request_json['config_id'], target_id=target_id, scanner_id=request_json['scanner_id']).xpath('@id')
            return {'UUID': task_id[0]}
        except GvmError as e:
            abort(500)
    elif request_json.keys() >= {"hosts", "config_id", "scanner_id"}:
        try:
            with Gmp(connection=connection, transform=transfrom) as gmp:
                gmp.authenticate(username, password)
                target_id = gmp.create_target(name=request_json['hosts'] + ' target', hosts=[request_json['hosts']], port_range='T:1-65535,U:1-65535').xpath('@id')
                task_id = gmp.create_task(name=request_json['hosts'] + ' task', config_id=request_json['config_id'], target_id=target_id, scanner_id=request_json['scanner_id']).xpath('@id')
            return {'UUID': task_id[0]}
        except GvmError as e:
            abort(500)
    else:
        abort(400)
```

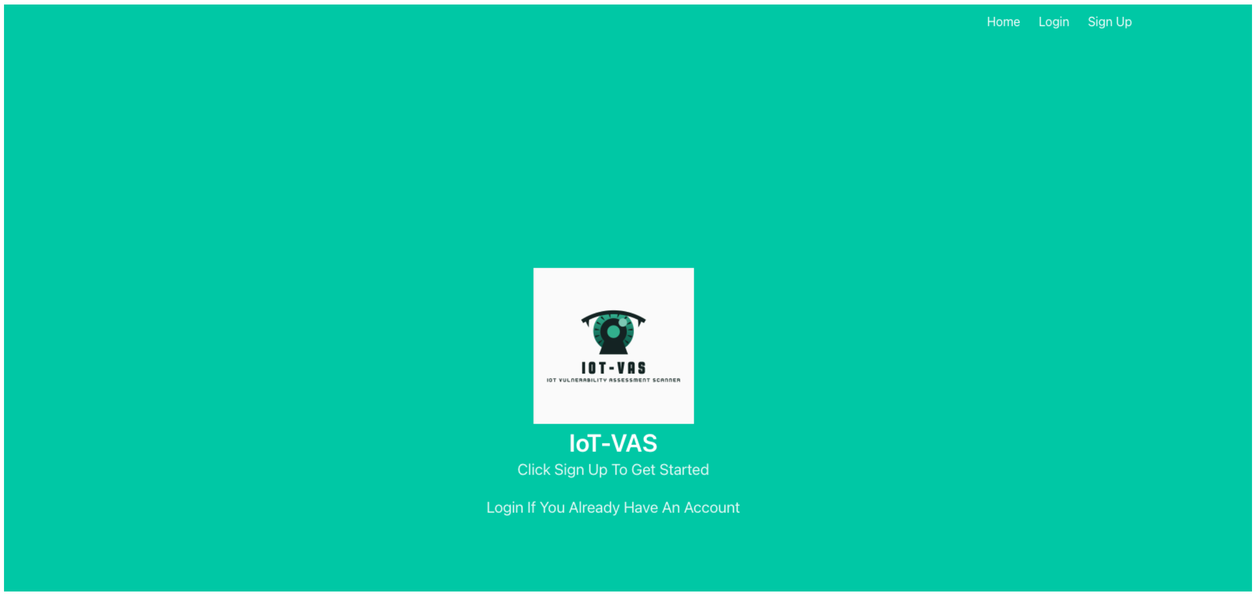
Python Create Task API Route

-The functions that require post requests to run (create task shown above) are much more complex than how the simple get request functions work (get_version, get_scanners and get_configs).



Web App User Interface

Spring Pictures

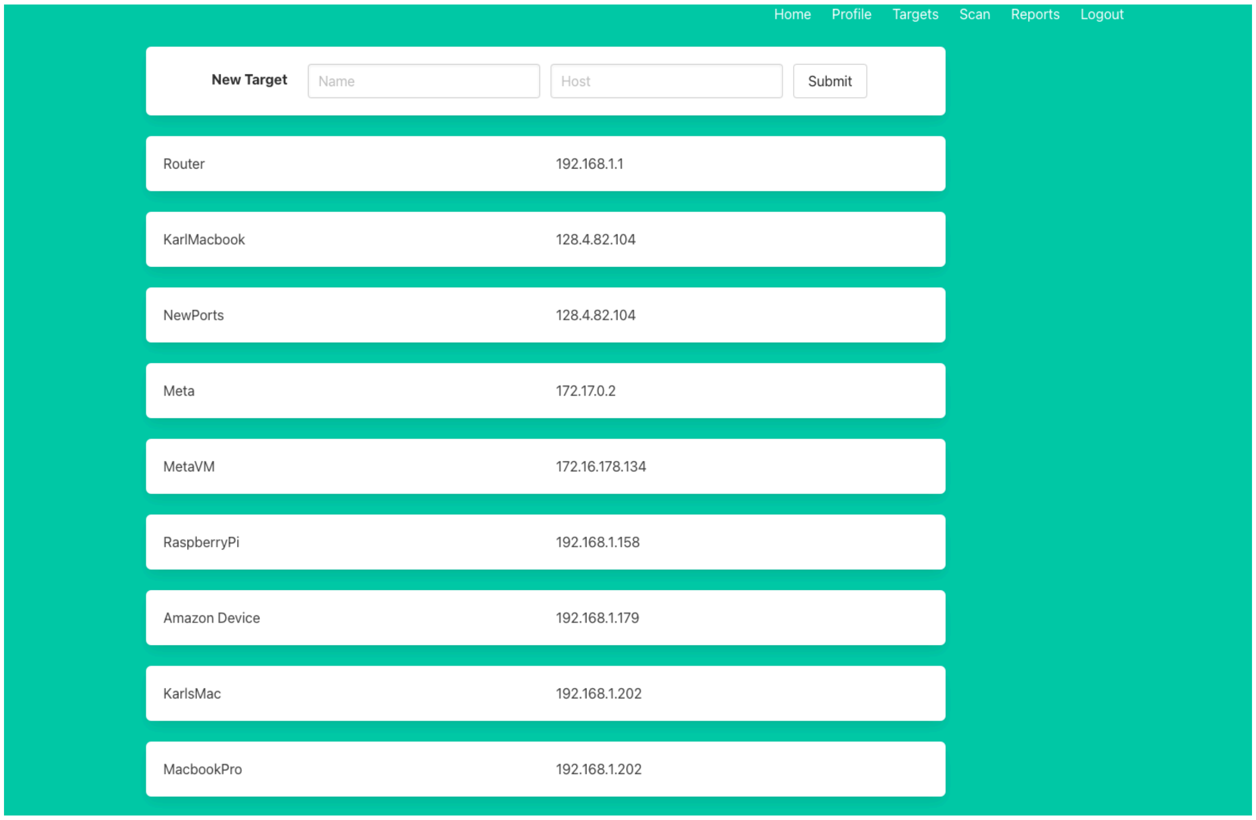


IoT-VAS

Click Sign Up To Get Started

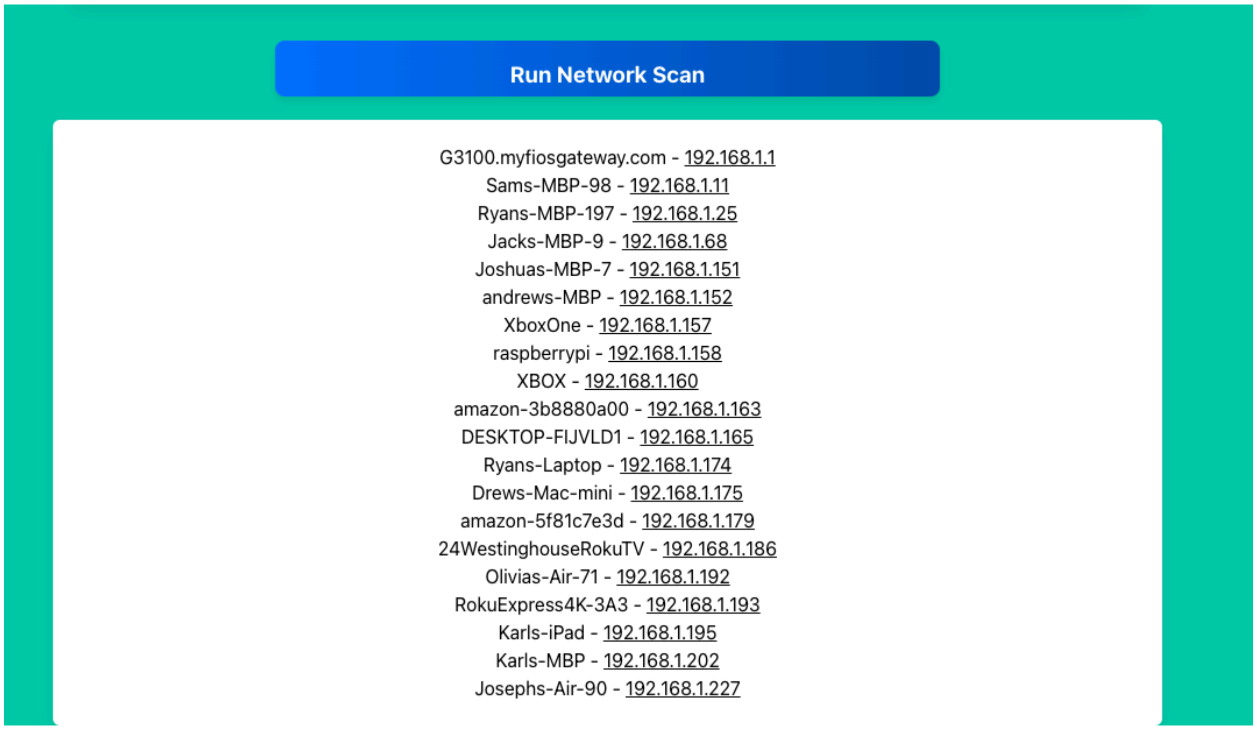
Login If You Already Have An Account

Web Application Homepage



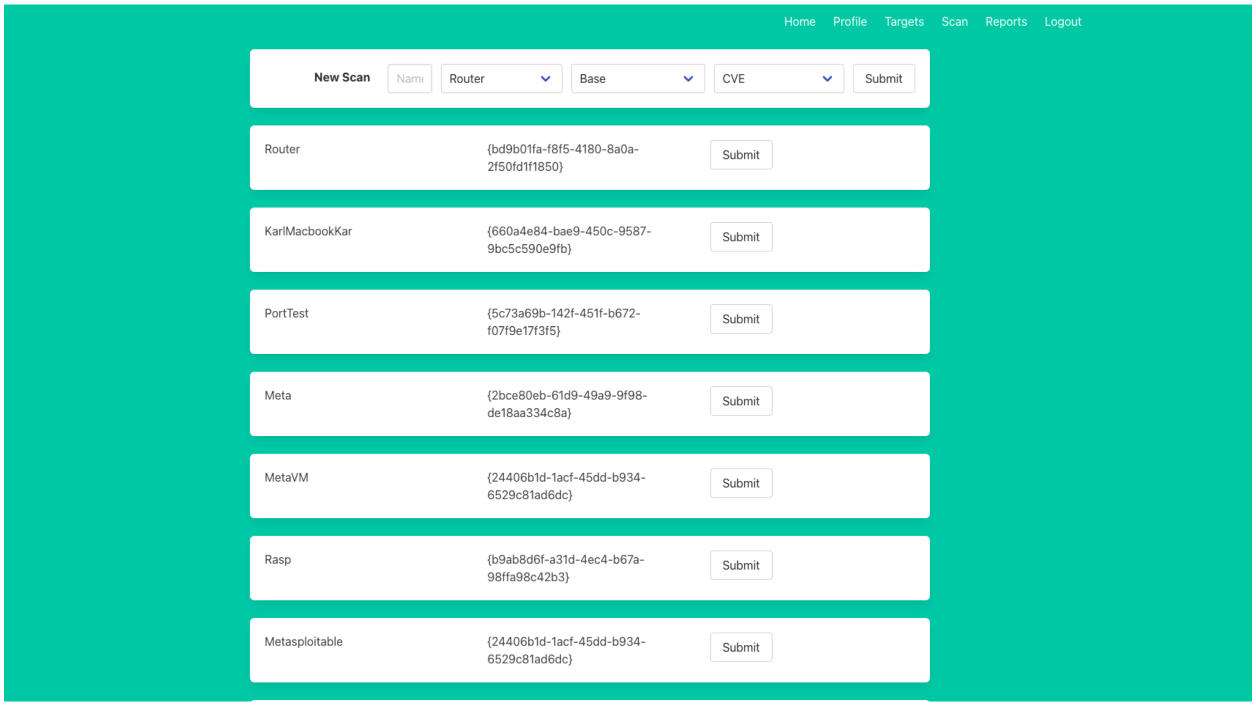
Router	192.168.1.1
KarlMacbook	128.4.82.104
NewPorts	128.4.82.104
Meta	172.17.0.2
MetaVM	172.16.178.134
RaspberryPi	192.168.1.158
Amazon Device	192.168.1.179
KarlsMac	192.168.1.202
MacbookPro	192.168.1.202

Target Creation Page



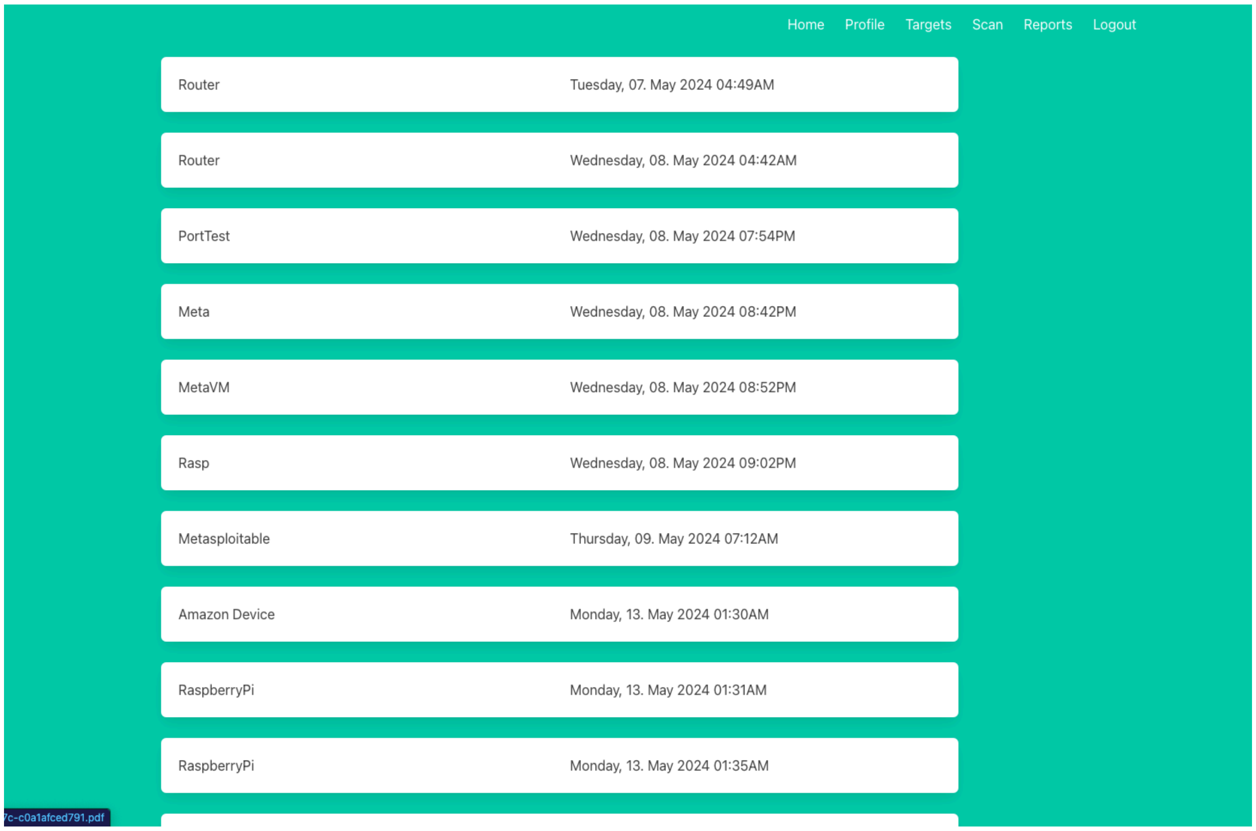
G3100.myfiosgateway.com - 192.168.1.1
Sams-MBP-98 - 192.168.1.11
Ryans-MBP-197 - 192.168.1.25
Jacks-MBP-9 - 192.168.1.68
Joshuas-MBP-7 - 192.168.1.151
andrews-MBP - 192.168.1.152
XboxOne - 192.168.1.157
raspberrypi - 192.168.1.158
XBOX - 192.168.1.160
amazon-3b8880a00 - 192.168.1.163
DESKTOP-FIJVLD1 - 192.168.1.165
Ryans-Laptop - 192.168.1.174
Drews-Mac-mini - 192.168.1.175
amazon-5f81c7e3d - 192.168.1.179
24WestinghouseRokuTV - 192.168.1.186
Olivias-Air-71 - 192.168.1.192
RokuExpress4K-3A3 - 192.168.1.193
Karls-iPad - 192.168.1.195
Karls-MBP - 192.168.1.202
Josephs-Air-90 - 192.168.1.227

Network Scan Results



Router	{bd9b01fa-f8f5-4180-8a0a-2f50fd1f1850}	Submit
KarlMacbookKar	{660a4e84-bae9-450c-9587-9bc5c590e9fb}	Submit
PortTest	{5c73a69b-142f-451f-b672-f07f9e17f3f5}	Submit
Meta	{2bce80eb-61d9-49a9-9f98-de18aa334c8a}	Submit
MetaVM	{24406b1d-1acf-45dd-b934-6529c81ad6dc}	Submit
Rasp	{b9ab8d6f-a31d-4ec4-b67a-98ffa98c42b3}	Submit
Metasploitable	{24406b1d-1acf-45dd-b934-6529c81ad6dc}	Submit

Scan Creation Page

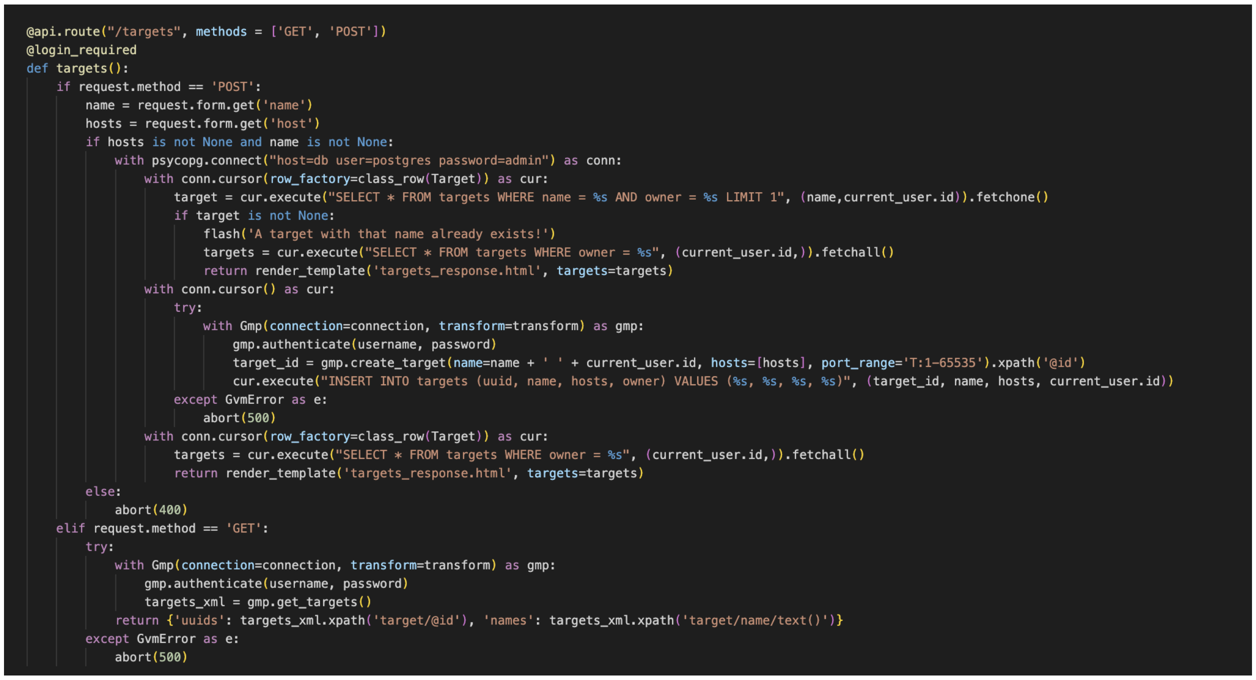


Home Profile Targets Scan Reports Logout

Router	Tuesday, 07. May 2024 04:49AM
Router	Wednesday, 08. May 2024 04:42AM
PortTest	Wednesday, 08. May 2024 07:54PM
Meta	Wednesday, 08. May 2024 08:42PM
MetaVM	Wednesday, 08. May 2024 08:52PM
Rasp	Wednesday, 08. May 2024 09:02PM
Metasploitable	Thursday, 09. May 2024 07:12AM
Amazon Device	Monday, 13. May 2024 01:30AM
RaspberryPi	Monday, 13. May 2024 01:31AM
RaspberryPi	Monday, 13. May 2024 01:35AM

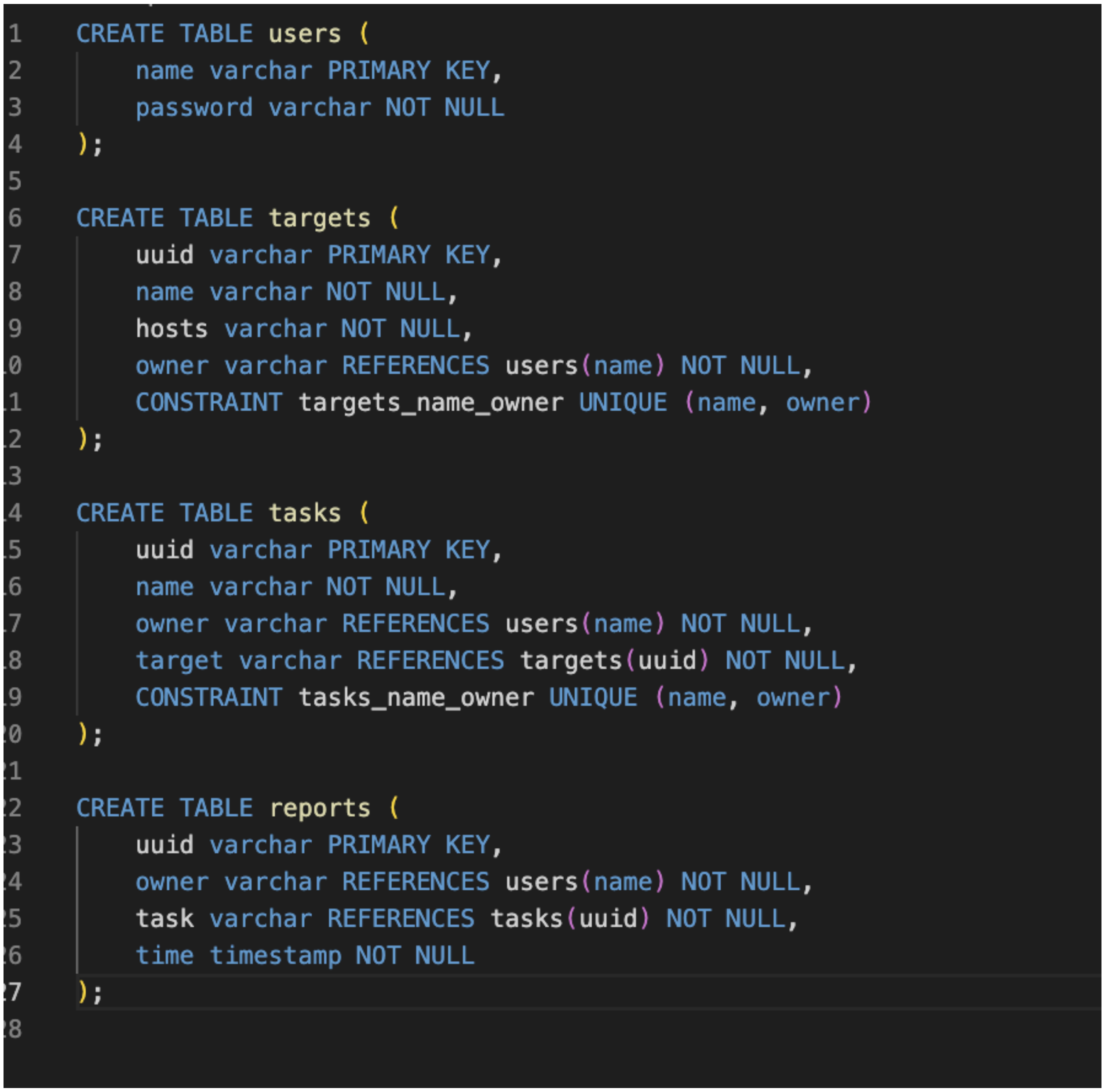
7c-c0a1afced791.pdf

Reports Page



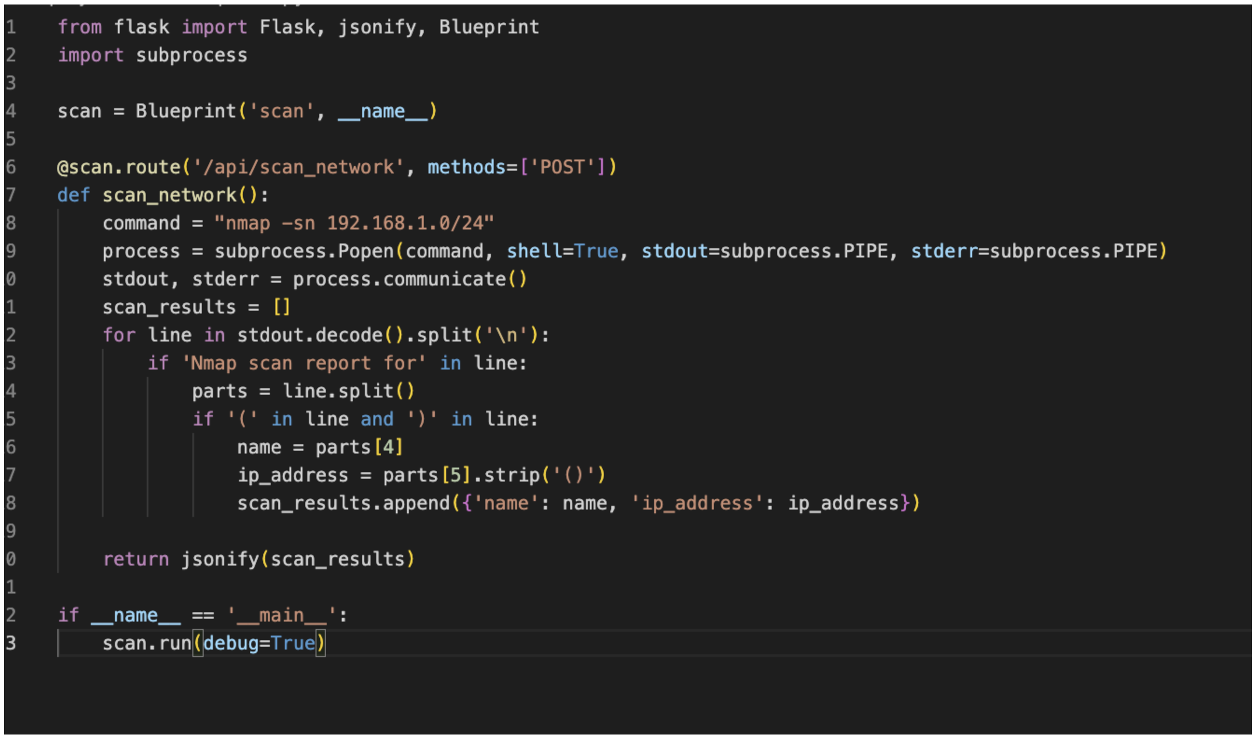
```python
@api.route("/targets", methods = ['GET', 'POST'])
@login_required
def targets():
    if request.method == 'POST':
        name = request.form.get('name')
        hosts = request.form.get('host')
        if hosts is not None and name is not None:
            with psycopg.connect("host=db user=postgres password=admin") as conn:
                with conn.cursor(row_factory=class_row(Target)) as cur:
                    target = cur.execute("SELECT * FROM targets WHERE name = %s AND owner = %s LIMIT 1", (name,current_user.id)).fetchone()
                    if target is not None:
                        flash('A target with that name already exists!')
                        targets = cur.execute("SELECT * FROM targets WHERE owner = %s", (current_user.id,)).fetchall()
                        return render_template('targets_response.html', targets=targets)
                with conn.cursor() as cur:
                    try:
                        with Gmp(connection=connection, transform=transform) as gmp:
                            gmp.authenticate(username, password)
                            target_id = gmp.create_target(name=name + ' ' + current_user.id, hosts=[hosts], port_range='T:1-65535').xpath('@id')
                            cur.execute("INSERT INTO targets (uuid, name, hosts, owner) VALUES (%s, %s, %s, %s)", (target_id, name, hosts, current_user.id))
                    except GvmError as e:
                        abort(500)
                with conn.cursor(row_factory=class_row(Target)) as cur:
                    targets = cur.execute("SELECT * FROM targets WHERE owner = %s", (current_user.id,)).fetchall()
                    return render_template('targets_response.html', targets=targets)
        else:
            abort(400)
    elif request.method == 'GET':
        try:
            with Gmp(connection=connection, transform=transform) as gmp:
                gmp.authenticate(username, password)
                targets_xml = gmp.get_targets()
            return {'uuids': targets_xml.xpath('target/@id'), 'names': targets_xml.xpath('target/name/text()')}
        except GvmError as e:
            abort(500)
```

Target Creation API



```sql
CREATE TABLE users (
    name varchar PRIMARY KEY,
    password varchar NOT NULL
);

CREATE TABLE targets (
    uuid varchar PRIMARY KEY,
    name varchar NOT NULL,
    hosts varchar NOT NULL,
    owner varchar REFERENCES users(name) NOT NULL,
    CONSTRAINT targets_name_owner UNIQUE (name, owner)
);

CREATE TABLE tasks (
    uuid varchar PRIMARY KEY,
    name varchar NOT NULL,
    owner varchar REFERENCES users(name) NOT NULL,
    target varchar REFERENCES targets(uuid) NOT NULL,
    CONSTRAINT tasks_name_owner UNIQUE (name, owner)
);

CREATE TABLE reports (
    uuid varchar PRIMARY KEY,
    owner varchar REFERENCES users(name) NOT NULL,
    task varchar REFERENCES tasks(uuid) NOT NULL,
    time timestamp NOT NULL
);
```

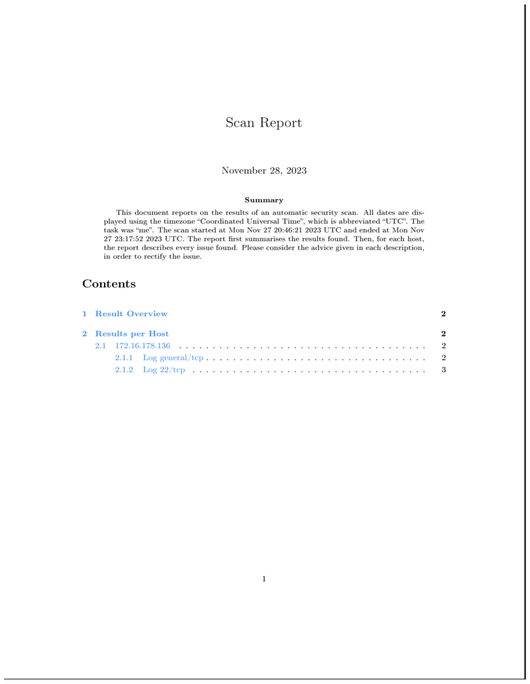
PostgreSQL Database Table Creation



```python
from flask import Flask, jsonify, Blueprint
import subprocess

scan = Blueprint('scan', __name__)

@scan.route('/api/scan_network', methods=['POST'])
def scan_network():
    command = "nmap -sn 192.168.1.0/24"
    process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    stdout, stderr = process.communicate()
    scan_results = []
    for line in stdout.decode().split('\n'):
        if 'Nmap scan report for' in line:
            parts = line.split()
            if '(' in line and ')' in line:
                name = parts[4]
                ip_address = parts[5].strip('()')
                scan_results.append({'name': name, 'ip_address': ip_address})

    return jsonify(scan_results)

if __name__ == '__main__':
    scan.run(debug=True)
```

Network Discovery API Script

Fall Results of Testing



Scan Report

November 28, 2023

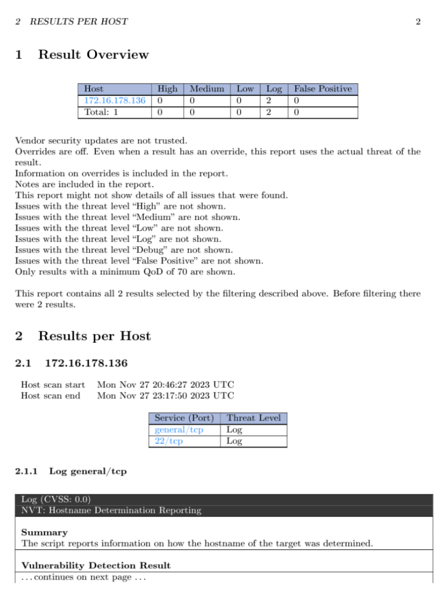
Summary

This document reports on the results of an automatic security scan. All dates are displayed using the timezone "Coordinated Universal Time", which is abbreviated "UTC". The task was "me". The scan started at Mon Nov 27 20:46:21 2023 UTC and ended at Mon Nov 27 23:17:52 2023 UTC. The report first summarises the results found. Then, for each host, the report describes every issue found. Please consider the advice given in each description, in order to rectify the issue.

Contents

1 Result Overview 2
2 Results per Host 2
2.1 172.16.178.136 2
2.1.1 Log general/tcp 2
2.1.2 Log 22/tcp 3

1



1 Result Overview

Host	High	Medium	Low	Log	False Positive
172.16.178.136	0	0	0	2	0
Total: 1	0	0	0	2	0

Vendor security updates are not trusted.
Overrides are off. Even when a result has an override, this report uses the actual threat of the result.
Information on overrides is included in the report.
Notes are included in the report.
This report might not show details of all issues that were found.
Issues with the threat level "High" are not shown.
Issues with the threat level "Medium" are not shown.
Issues with the threat level "Low" are not shown.
Issues with the threat level "Log" are not shown.
Issues with the threat level "Debug" are not shown.
Issues with the threat level "False Positive" are not shown.
Only results with a minimum QoD of 70 are shown.

This report contains all 2 results selected by the filtering described above. Before filtering there were 2 results.

2 Results per Host

2.1 172.16.178.136

Host scan start Mon Nov 27 20:46:27 2023 UTC
Host scan end Mon Nov 27 23:17:50 2023 UTC

Service (Port)	Threat Level
general/tcp	Log
22/tcp	Log

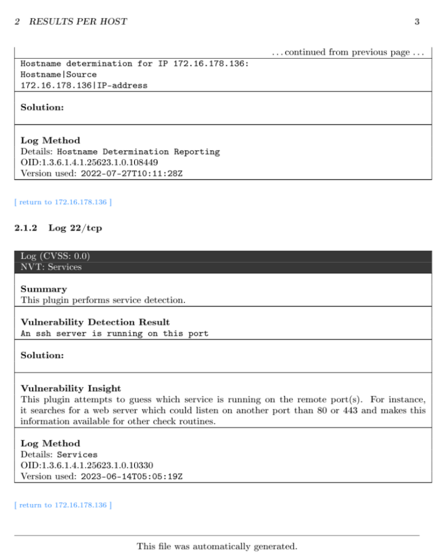
2.1.1 Log general/tcp

Log (CVSS: 0.0)
NVT: Hostname Determination Reporting

Summary
The script reports information on how the hostname of the target was determined.

Vulnerability Detection Result

. . . continues on next page . . .



. . . continued from previous page . . .

```
Hostname determination for IP 172.16.178.136:
Hostname|Source
172.16.178.136|IP-address
```

Solution:

Log Method
Details: `Hostname Determination Reporting`
OID:1.3.6.1.4.1.25623.1.0.108449
Version used: `2022-07-27T10:11:28Z`

[return to 172.16.178.136]

2.1.2 Log 22/tcp

Log (CVSS: 0.0)
NVT: Services

Summary
This plugin performs service detection.

Vulnerability Detection Result
`An ssh server is running on this port`

Solution:

Vulnerability Insight
This plugin attempts to guess which service is running on the remote port(s). For instance, it searches for a web server which could listen on another port than 80 or 443 and makes this information available for other check routines.

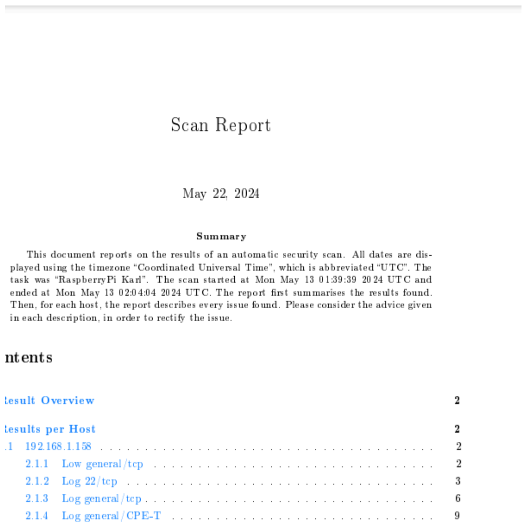
Log Method
Details: `Services`
OID:1.3.6.1.4.1.25623.1.0.10330
Version used: `2023-06-14T05:05:19Z`

[return to 172.16.178.136]

This file was automatically generated.

-After we were able to get the create target, create task, start task and get report functions working this is the result of one of our test scans.

Spring Results of Testing:



Scan Report

May 22, 2024

Summary

This document reports on the results of an automatic security scan. All dates are displayed using the timezone "Coordinated Universal Time", which is abbreviated "UTC". The task was "RaspberryPi Karl". The scan started at Mon May 13 01:39:39 2024 UTC and ended at Mon May 13 02:04:04 2024 UTC. The report first summarises the results found. Then, for each host, the report describes every issue found. Please consider the advice given in each description, in order to rectify the issue.

ntents

tesult Overview 2

tesults per Host 2

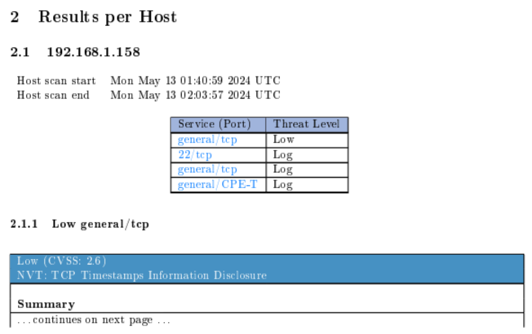
.1 192.168.1.158 2

2.1.1 Low general/tcp 2

2.1.2 Log 22/tcp 3

2.1.3 Log general/tcp 6

2.1.4 Log general/CPE-T 9



2 Results per Host

2.1 192.168.1.158

Host scan start	Mon May 13 01:40:59 2024 UTC
Host scan end	Mon May 13 02:03:57 2024 UTC

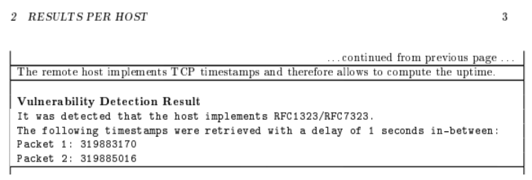
Service (Port)	Threat Level
general/tcp	Low
22/tcp	Log
general/tcp	Log
general/CPE-T	Log

2.1.1 Low general/tcp

Low (CVSS: 2.6)
NVT: TCP Timestamps Information Disclosure

Summary

. . . continues on next page . . .

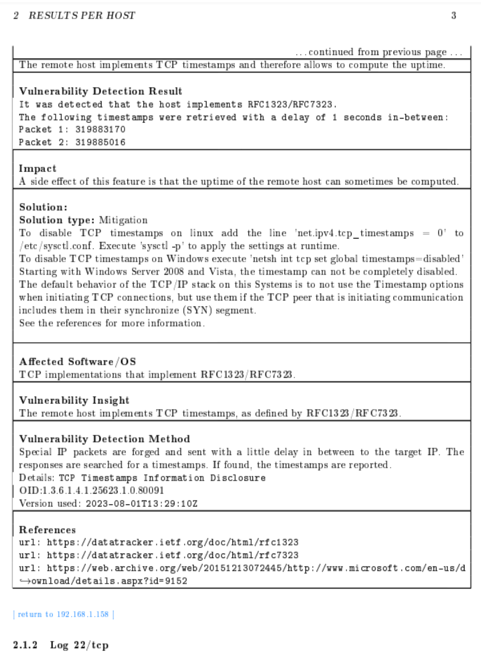


. . . continued from previous page . . .

The remote host implements TCP timestamps and therefore allows to compute the uptime.

Vulnerability Detection Result

```
It was detected that the host implements RFC1323/RFC7323.
The following timestamps were retrieved with a delay of 1 seconds in-between:
Packet 1: 319883170
Packet 2: 319885016
```



. . . continued from previous page . . .

The remote host implements TCP timestamps and therefore allows to compute the uptime.

Vulnerability Detection Result

```
It was detected that the host implements RFC1323/RFC7323.
The following timestamps were retrieved with a delay of 1 seconds in-between:
Packet 1: 319883170
Packet 2: 319885016
```

Impact

A side effect of this feature is that the uptime of the remote host can sometimes be computed.

Solution:

Solution type: Mitigation

To disable TCP timestamps on linux add the line 'net.ipv4.tcp_timestamps = 0' to /etc/sysctl.conf. Execute 'sysctl -p' to apply the settings at runtime.

To disable TCP timestamps on Windows execute 'netsh int tcp set global timestamps=disabled'

Starting with Windows Server 2008 and Vista, the timestamp can not be completely disabled.

The default behavior of the TCP/IP stack on this Systems is to not use the Timestamp options when initiating TCP connections, but use them if the TCP peer that is initiating communication includes them in their synchronize (SYN) segment.

See the references for more information.

Affected Software/OS

TCP implementations that implement RFC1323/RFC7323.

Vulnerability Insight

The remote host implements TCP timestamps, as defined by RFC1323/RFC7323.

Vulnerability Detection Method

Special IP packets are forged and sent with a little delay in between to the target IP. The responses are searched for a timestamps. If found, the timestamps are reported.

Details: `TCP Timestamps Information Disclosure`

OID:1.3.6.1.4.1.25623.1.0.80091

Version used: `2023-08-01T13:29:10Z`

References

```
url: https://datatracker.ietf.org/doc/html/rfc1323
url: https://datatracker.ietf.org/doc/html/rfc7323
url: https://web.archive.org/web/20151213072445/http://www.microsoft.com/en-us/d
↪ownload/details.aspx?id=9152
```

[return to 192.168.1.158]

2.1.2 Log 22/tcp

Scan Results From Raspberry Pi (Date is Incorrect)

Project Results:

One of the things that we have learned so far working on this project is using OpenVAS to scan for vulnerabilities. Specifically interacting with OpenVAS through the Greenbone Management Protocol and its accompanying Python library, python-gvm. More generally we learned how to use the documentation for a piece of software to interact with it programmatically. We also learned how to use docker to run existing containers, such as the Greenbone community containers used as a way to distribute OpenVAS, but also build our own containers like our flask server and our nginx server. We also learned to use docker compose to manage multiple containers and the connections between them through a yaml file that described your setup and simplifies setting everything up and resetting everything if need be. We gained familiarity with using the flask web framework to build web applications in Python. Neither of us were previously familiar with creating full stack web applications either, so coordinating both the frontend and the backend to work together was something we had never done before. We also learned to use NGINX as a reverse proxy and for serving static content.

In terms of future work, we are definitely going to start by polishing up the front end at least a bit now that we have the minimal functionality working. Then, to improve the overall functionality of the application, we need to segment user data on the backend so that each user can store and only access their own report, scans, and targets. Part of that work is going to involve improving the user management by restructuring the database to store more than just username and password hashes. We then need to look into deploying this to an actual hosting provider so that it can be used from anywhere. We'll have to work out the details of how we are going to limit what IPs people can scan because we need to prove they own them. Right now the idea is having them serve a randomly generated token on a random port to prove they own the machine, although there might be flaws in that approach. If we just let anyone put in any IP, then anyone could essentially launch attacks on behalf of us directed towards anyone and that could get us in trouble. There is also an issue we need to resolve pertaining to multiple clients being connected to the OpenVAS server at the same time and hopefully we can come up with a scalable solution.

In the spring, we focused on expanding the functionality of our application and enhancing its usability. One of the major additions was the network discovery feature, which automatically uses nmap and a Python library called subprocess to run terminal commands. This feature scans the network, parses the output, and displays the device IPs and names, creating targets for further scanning. Implementing this feature was challenging as it required ensuring that the subprocess commands executed correctly within our Docker environment and accurately parsing the nmap output to display it on the web app. This addition greatly improved the user experience by automating the initial steps of identifying devices for scanning.

We also integrated Postgres databases to manage user logins, targets, tasks (scans), and report PDFs. This required setting up a robust database schema and ensuring seamless interaction between the web app and the database. We wrote SQL scripts to handle data insertion, retrieval, and updates efficiently. Managing data consistency and integrity, especially with multiple users interacting with the application simultaneously, was a significant challenge. We implemented proper error handling and validation checks to ensure the data remained accurate and reliable. This database integration allowed us to store and display information more effectively, improving the overall functionality of the web app.

To improve the user interface, we redesigned the web application to create dedicated pages for tasks, targets, and reports instead of consolidating everything on one page. This modular approach made navigation more intuitive and allowed users to manage scans and view results more easily. Each page was developed using Flask and Jinja templates, ensuring consistency and ease of use across the application. This design change significantly enhanced the user experience by making the application more organized and user-friendly.

Throughout the spring, testing and debugging were critical to ensure the new features' reliability and functionality. We used Adminer, a lightweight tool for managing databases, to inspect data directly and verify that requests from the web app were correctly added to the Postgres database. Adminer helped us identify whether issues were due to the web app not properly adding data to the database tables or if the web app was incorrectly pulling information from the tables. This clear view of the data was instrumental in troubleshooting and maintaining the application's reliability.
Additionally, we made significant adjustments to the python-gvm API to better suit our web application's needs. While the python-gvm API provided a robust interface for interacting with OpenVAS, we needed it to seamlessly integrate with our Postgres databases. To achieve this, we included SQL commands within the API functions to handle data operations directly. This allowed us to push information to the database when new scans were initiated and pull information from the database to display results and updates to the users. Integrating SQL commands into the API streamlined our data management process, reducing the need for separate database handling code and ensuring that all interactions with the scanner and the database were tightly coupled. For example, when a new scan target was created, the API not only communicated with OpenVAS to set up the target but also executed SQL commands to store the target details in the Postgres database. Similarly, when retrieving scan results, the API pulled

the relevant data from the database, ensuring that users had up-to-date information available on the web application. This approach minimized data redundancy and improved the efficiency of our application. By embedding SQL commands within the API, we maintained a consistent flow of data between the scanner, the database, and the frontend, enhancing the overall reliability and performance of our system.
By the end of the spring, these enhancements had significantly improved our project's functionality and user experience. The network discovery feature allowed users to easily identify and target devices on their network for scanning. The integration of Postgres databases provided a robust and scalable solution for managing user data and scan results. The redesigned user interface made it easier for users to navigate the application and manage their scans effectively. Overall, these improvements brought us closer to realizing our vision for a comprehensive and user-friendly web application for IoT vulnerability scanning.

Engineering Standards and Constraints:
CFAA: Must have authorization from a system administrator or owner before running a vulnerability scanner on a computer that you do not own

ECMA-262: Defines the ECMAScript general-purpose programming language, used for javascript.

ECMA-404: Defines JSON, a lightweight, text-based, language-independent syntax for defining data interchange formats. Used to pass information back and forth through our API.

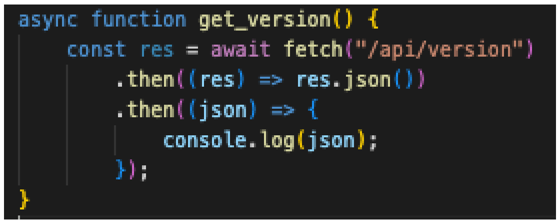
PEP 241: Defines metadata for python packages, used to fetch the python packages we used from PyPI

IEEE 1003.1-2017: Defines a standard operating system interface and environment, used as the platform to run our docker containers and

ISO 8879: Defines a standard generalized markup language. Used as the interface between our flask server and OpenVAS.

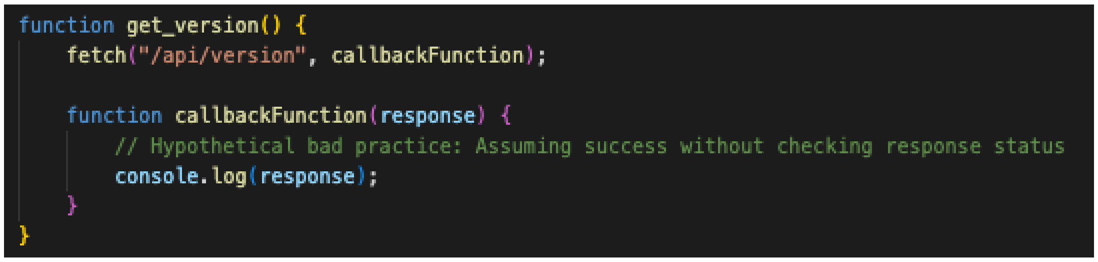
CFAA-1030: Must have authorization from a system administrator or owner before running a vulnerability scanner on a computer that you do not own. This may require a signed document with proof of permission to scan a machine that you do not own. As of right now we don’t need this because we have only used it on our own personal machines. When we come to the point of testing on other users' machines we will make sure to have written permission for running our software on their machine.

ECMA-262: Defines the ECMAScript general-purpose programming language, used for javascript. We must implement the best practices for JavaScript so our application runs smoothly and is secure. We must regularly update our JavaScript codebase to leverage new features and improve syntax while improvements are introduced.



```
async function get_version() {
    const res = await fetch("/api/version")
        .then((res) => res.json())
        .then((json) => {
            console.log(json);
        });
}
```

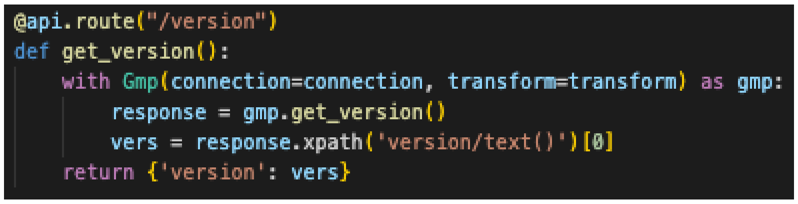
The use of modern JavaScript features like async/await for asynchronous operations demonstrates adherence to ECMAScript standards, making code more readable and easier to maintain.



```
function get_version() {
    fetch("/api/version", callbackFunction);

    function callbackFunction(response) {
        // Hypothetical bad practice: Assuming success without checking response status
        console.log(response);
    }
}
```

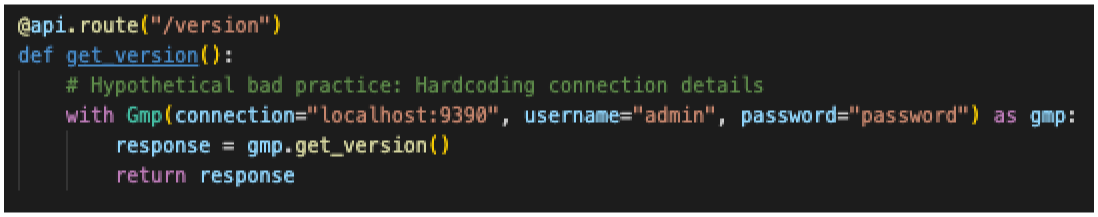
This example uses outdated callback patterns, lacks error handling, and does not check the response's status, leading to potential issues in understanding the fetch operation's outcome.

PEP-8: Style Guide for Python Code, which provides the coding conventions for writing legible Python code. This also requires us to follow good practices of python code meaning that we must keep sensitive host and user information secret.



```
@api.route("/version")
def get_version():
    with Gmp(connection=connection, transform=transform) as gmp:
        response = gmp.get_version()
        vers = response.xpath('version/text()')[0]
    return {'version': vers}
```

This is our code for getting the scanner version which follows the best practices by handling API requests in a structured manner.



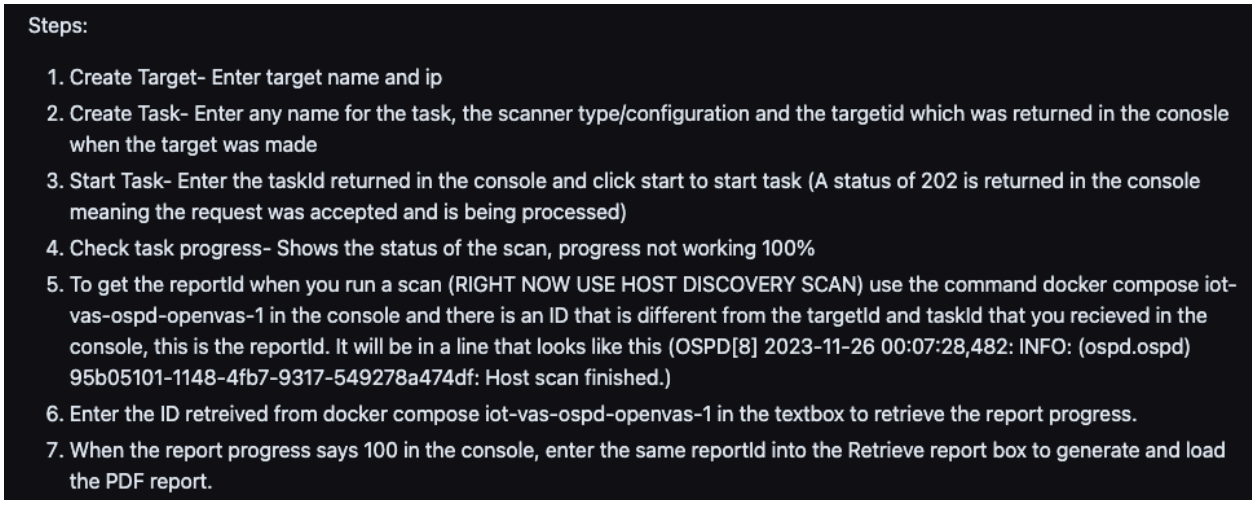
```python
@api.route("/version")
def get_version():
    # Hypothetical bad practice: Hardcoding connection details
    with Gmp(connection="localhost:9390", username="admin", password="password") as gmp:
        response = gmp.get_version()
        return response
```

Here is an example of what is considered to be bad practice. This code is hardcoding sensitive details such as connection information and credentials. The code also lacks error handling and access to response data which can lead to security vulnerabilities and poor data handling.

NIST-SP-800-115 and OWASP Testing Guide: Technical guide for information and security testing, this gives us guidelines to follow while we conduct our security testing and vulnerability scanning. This involves identifying which systems, networks or applications will be scanned and the types of vulnerabilities we are looking to identify. Thor requires us to run commands such as ifconfig so we know for sure that we are running a scan on the correct IP and not on someone else's device by accident.

The OWASP testing guide shows how to properly secure our web application and helps us avoid common web exploits that can be used to breach a user's private information. These exploits include things such as brute forcing which would allow someone to force their way into another user's account. In order to combat against this vulnerability we can only allow users a certain amount of attempts at entering the password in order to prevent someone from trying every combination with a brute force script.

IEEE 829: Provides a standard for us to document all aspects of testing, from planning to reporting outcomes and the resources required for testing our software. The software must have documentation describing the steps in order for the application to run allowing a new user to easily follow. We documented exactly how our application functions and included the necessary steps to successfully get the application run as intended.


Steps:

1. Create Target- Enter target name and ip
2. Create Task- Enter any name for the task, the scanner type/configuration and the targetid which was returned in the conosle when the target was made
3. Start Task- Enter the taskId returned in the console and click start to start task (A status of 202 is returned in the console meaning the request was accepted and is being processed)
4. Check task progress- Shows the status of the scan, progress not working 100%
5. To get the reportId when you run a scan (RIGHT NOW USE HOST DISCOVERY SCAN) use the command docker compose iot-vas-ospd-openvas-1 in the console and there is an ID that is different from the targetId and taskId that you recieved in the console, this is the reportId. It will be in a line that looks like this (OSPD[8] 2023-11-26 00:07:28,482: INFO: (ospd.ospd) 95b05101-1148-4fb7-9317-549278a474df: Host scan finished.)
6. Enter the ID retreived from docker compose iot-vas-ospd-openvas-1 in the textbox to retrieve the report progress.
7. When the report progress says 100 in the console, enter the same reportId into the Retrieve report box to generate and load the PDF report.

References:

https://ecma-international.org/publications-and-standards/standards/ecma-262/
https://ecma-international.org/publications-and-standards/standards/ecma-404/
https://peps.python.org/pep-0241/
https://www.nacdl.org/Landing/ComputerFraudandAbuseAct

Appendix:

Equipment used:

- Docker
  - Greenbone community containers
  - NGINX container
    - Static files
    - Config for reverse proxying
  - Python container
    - Flask server code
    - Templates
    - SQLite database volume
    - requirements.txt file for installing libraries
- Docker compose
- Metasploitable (for testing)
- VirtualBox (for running Metasploitable)

Budget:

- $0 so far
- Might need to allocate some money to host our app in the cloud in the future although this can often be done for free as long as traffic is low enough