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 These vulnerabilities can come from things like weak authentication, outdated software/firmware and insecure communication across networks. 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.

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.

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.

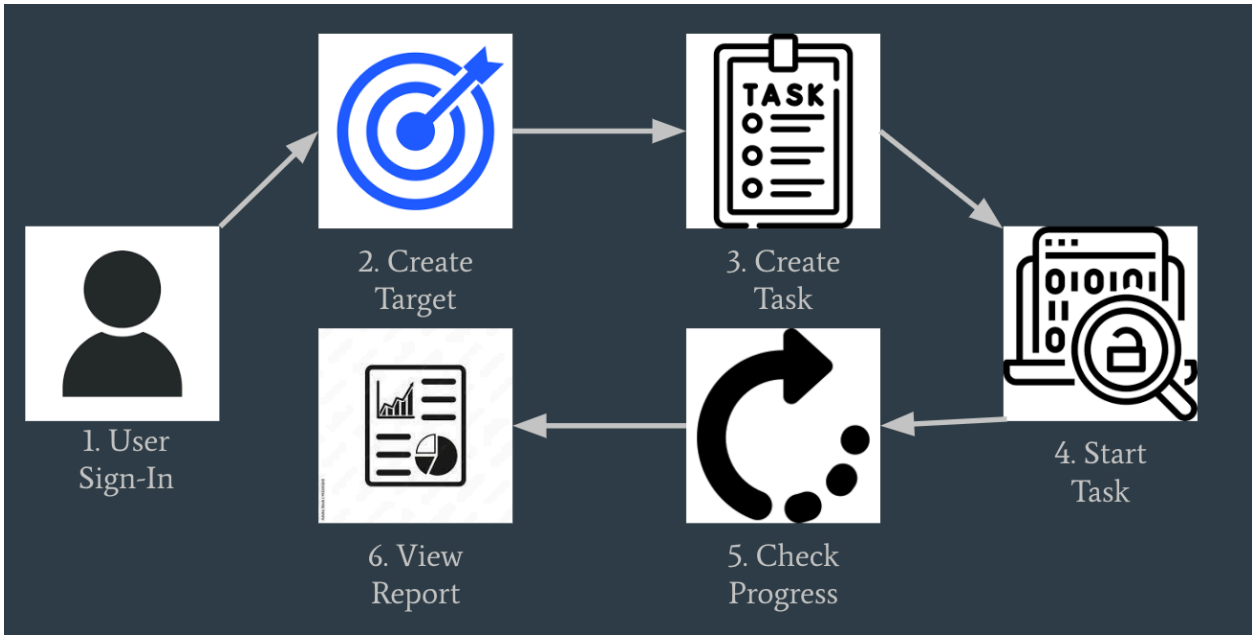
Understanding was only part of the problem, the next problem was how we were going to actually accomplish this. Overcoming this problem was a slow learning process. Doing a lot of research we decided that using docker would be the easiest way to host our frontend and backend to allow them to communicate. 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.

Design:

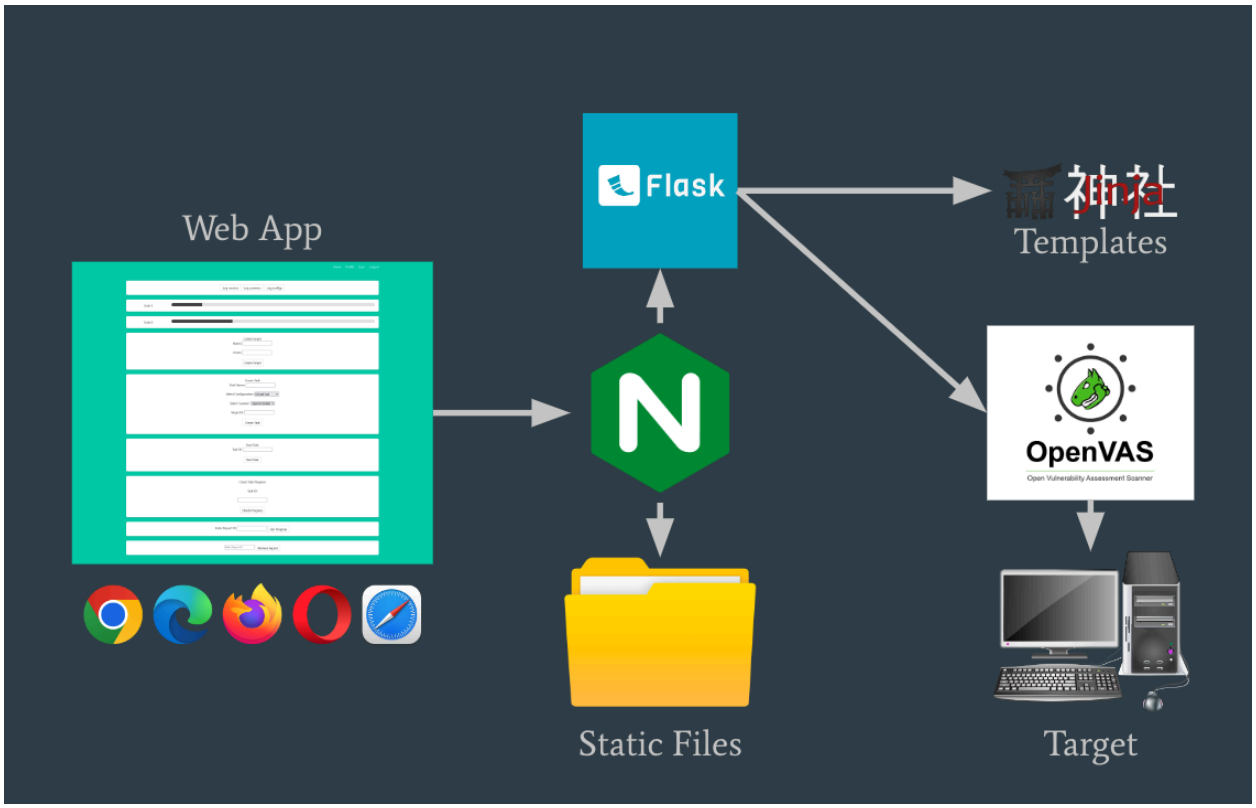
The overall design of our project includes a backend and a frontend. The backend is for our database, server, javascript functions and our API and our frontend is what the user will see and interact with in order to run scans. Docker is the application that ties this all together, allowing for them to 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.



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);
        });
}
```

Javascript for Web App

```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)
```

Python API Functions

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

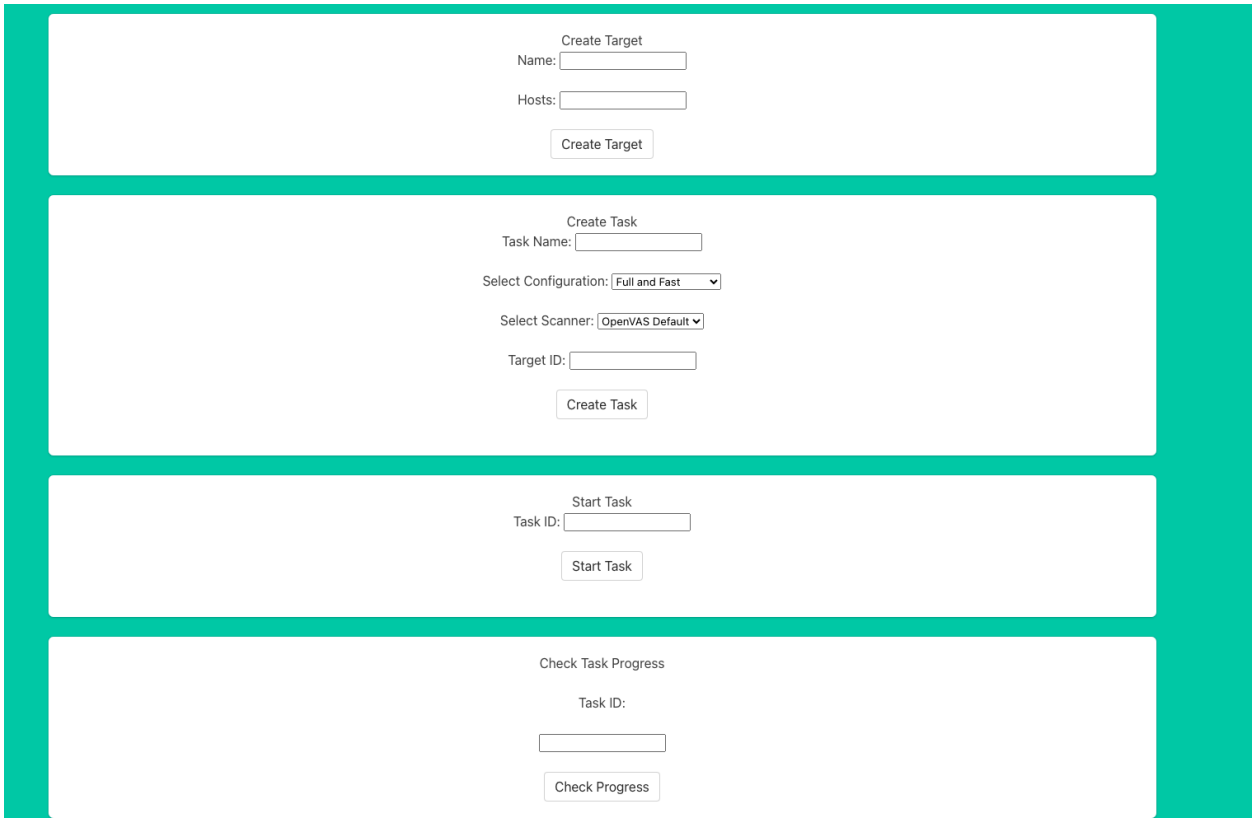
```javascript
async function createTask() {
    const taskName = document.getElementById('taskName').value;
    const configId = document.getElementById('configId').value;
    const scannerId = document.getElementById('scannerId').value;
    const targetId = document.getElementById('targetId').value;

    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

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

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]

-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.

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.

Engineering Standards:

- 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.

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