

CYBR410 Group Project: Big Deployment

Paul Aguilar
Front-End

Nehemiah Fiedls
Logging

Eric Leachman
Orchestration

Thomas Longwell
Defense

Robert Rutherford
Defense

Spencer West
Services

Nicholas Burlakov
Services

Abstract

The goal of this project is to work as a group with each member implementing a different aspect making up a larger deployment. The end goal is a secure and easily accessible web server with an aesthetic front end, useful and easy to use services, a robust defense to deter attackers, and a method to log all traffic on the server.

1 Introduction

This project is designed to have groupmates work together to design and implement a full deployment. Each team mate was assigned a specific piece of the deployment as their responsibility and each section of this paper relates to a section of the deployment. Front-End is in charge of creating the user interface of the deployment as well as the web server that will host the services. The Services team then implemented an URL to physical location feature as well as a URL to current weather forecast converter. Behind the scenes the Defense team integrated an IP blocking honeypot to deter attacks, while Logging used a Sysdig script to keep a record of all traffic. This was handed to Orchestration, who spun up the servers on the designated IP assigned by the professor.

2 Orchestration

The orchestration for this deployment concerns itself with two key areas. The first area, services, have been designed utilizing docker containers and docker compose for simple build up and tear down. Our docker compose solution utilizes two docker images, one running NGINX as a reverse proxy, and the other running a flask server locally which serves as an endpoint for the NGINX server.

The second area, defense, utilizes OpenCanary and IPTables. While OpenCanary is also a docker container, the complexity in building it warrants its own separate orchestration. To accomplish this, we utilize some basic bash scripting to automate the necessary commands to run OpenCanary with the

configuration and services required. We also include IPTables commands part of this bash script. These rules filter ports and drop traffic that attempts to enumerate OpenCanary and its ports.

By automating most of the more complex tasks, our services and defenses can be deployed quickly and by a greater number of individuals, even those not intimately familiar with its inner workings. The use of docker containers and docker compose also allow our deployment to be mobile, only requiring prospective users to install a small number of programs (i.e., docker, docker compose, git).

3 Front end

The front end is designed around a few simple HTML template pages, stylized with a CSS style guide. There are three main templates and a variation on two – the first template is the index, a home page that is repeatedly linked on some blank links in the navigation bar. There are also two pages dedicated to the address and weather API's. These pages have two versions, a search and an output template. The search template uses a simple form for the user to input a URL. The output template displays the results from the API call. These files are hosted via a flask server. [1]

The flask server uses the render_template and request objects to function seamlessly with the HTML and CSS. Following traditional directory organizations, a templates and static directory are used to hold those files for flask to access.

```
/app
- fServer.py
/templates
- index.html
/static
logo png's
/fonts
/styles
- style.css
```

Flask's `render_template` is used to display the correct HTML template in any given scenario, either from the `hostWeather` method or directly in the HTML links. The request object is used to handle the form submissions so that the user provided URL's can be used by the API methods to generate the correct output. Both `hostAddress` and `hostWeather` use a similar method to pass the URL correctly depending on whether a POST request was made from a form submission, via requests `request.form` method or directly from the URL query string. From there the API methods are used to gather the data, which is the outputted using the `render_template` method and displayed with in the output template.

Listing 1: `hostWeather` method handles GET/POST requests

```

1 @app.route("/weather", methods=['GET', '
2     POST'])
3 def hostWeather():
4     if request.method == 'POST':
5         # Logic to handle form
6         # submission via POST request
7         incomingHost = request.form['url
8     '']
9     else:
10        # Logic to handle GET request
11        incomingHost = request.args.get(
12            'url', '')
13
14    ipAddress = getIP(incomingHost)
15
16    if checkCache(cacheW, incomingHost):
17        weather = cacheW[incomingHost]
18    else:
19        weather=getWeather(incomingHost)
20        addCache(cacheW, incomingHost,
21            weather)

```

While the host methods route to the correct output page, the remaining routes direct a user to the index or the search pages. The following sequence diagram shows the basic user interaction with the site:

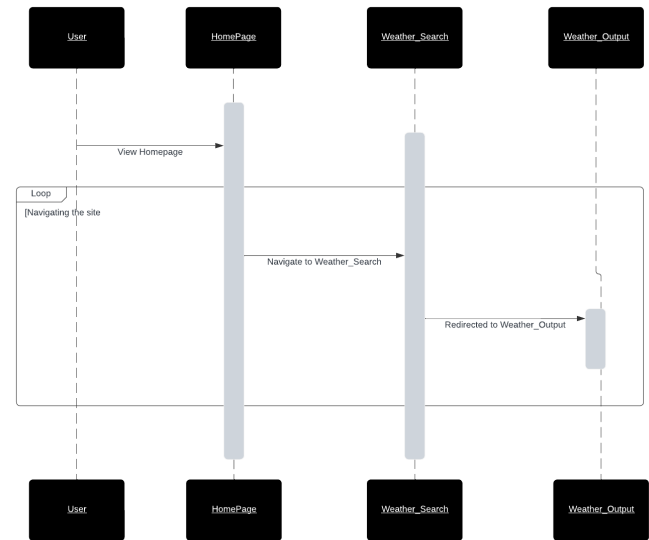


Figure 1: Environment Diagram

The first method that is used is `getIP`, which first strips the incoming URL of any whitespace and then uses the socket library's `gethostbyname` method to save the IP. This method is then used by both `getWeather` and `getAddress` in order to find the related information. `getAddress` uses the `subprocess` `getstatusoutput` method to run the `whois` commandline tool to search for the registrar information based on the provided IP. It then splits the incoming tuple by a newline and parses out the different address fields, combines them and returns a whole address.

The `getWeather` method then passes that address in to a geocoding API to find the latitude and longitude of the address, which is required by the weather API that is used. The geocoding call returns a json file, that needs to be parsed and split between x and y coordinates correctly. The request's `get` method is then used to make an API call to `weather.gov` [2] via the command line again. This outputs another json file that is then parsed to find the correct URL that contains the weather information. The `get` method is used once again on that forecast URL, which outputs the final json file, this is correctly parsed and returned as the weather for the given IP.

4 Services

This web application incorporates two API's in the flask server. One service takes a URL and outputs the address of the registered owner based on the whois registry. The other uses that address and another weather API to output the weather of that given location. It also uses a list to cache the data of visited sites, to improve performance. To do all this, the API is broken up in to multiple methods, which ultimately called by the `hostWeather` and `hostAddress` methods when a form submission is made on the webpage.

Listing 2: `getWeather` method

```

1 def getWeather(incoming):
2
3
4
5     ipAddress= getIP(incoming)
6
7     addy = getAddress(ipAddress)
8
9     geocodingApi="https://geocoding.geo.

```

```

        census.gov/geocoder/locations/
        address?street="+addy[0]+"&city=
        "+addy[1]+"&state="+addy[2]+"&
        zip="+addy[3]+"&benchmark=
        Public_AR_Current&format=json"
10    georesponse=requests.get(
        geocodingApi)
11    js = json.loads(georesponse.text)
12
13    outputx=js['result']['addressMatches
        '][0]['coordinates']['x']
14    outputy=js['result']['addressMatches
        '][0]['coordinates']['y']
15    lat=format(outputy)
16    lon=format(outputx)
17
18
19    # base API string for weather.gov
20    weather_s = "https://api.weather.gov
        /points/"
21
22    # use the commandline input and the
        weather_s to make API call
23    response = requests.get(weather_s+
        lat+", "+lon)
24
25    # convert it to json
26    js = json.loads(response.text)
27
28    # find the forecast URL based on the
        API page
29    forecast_URL = js['properties']['
        forecast']
30
31    # call the API again to get
        theforecast
32    final_response = requests.get(
        forecast_URL)
33
34    #parse json
35    js = json.loads(final_response.text)
36
37    #print the forecast
38    weather=(js['properties']['periods'
        ][0]['detailedForecast'])
39
40    return(weather)

```

The remaining methods handle the caching, either directly adding to it based on the correct list or traversing it to search for the given key which returns true or false. The host methods use this to first check the cache and then output the value based on the given key or add the key/value pair to the given cache if it isn't found. To summarize, this script creates a Flask

web server that provides endpoints for IP address lookup and weather information retrieval, caching the results to improve performance. It also demonstrates how to integrate external APIs and system commands into a Flask application.

5 Logging

The bash script I've made automates the monitoring of a system by capturing system activity using the sysdig tool every 30 minutes. Each session stops when the data file reaches about 20 MB or after 4 hours of total runtime. The script checks if each generated file exceeds a size limit of 20MB and, if so, runs a specific program. This can be useful for system monitoring, security analysis, and troubleshooting. The script helps manage data capture efficiently but requires careful consideration regarding storage, privacy, and the potential impact on system performance.

6 Defense

In this section we will discuss the implementation of defense or our group project in detail. For our groups approach to defense we used a two part system built of a honeypot and a script to control traffic. The goal of the honeypot is to provide a avenue for traffic that deviates from the norm. Once this deviated or unauthorized traffic is found the script uses iptables to block incoming traffic effectively removing traffic from unauthorized users, such as those who would scan the server or try to connect to a fake service on the honey pot. This defensive approach assumes authorized users will know what is in scope and not try to connect to or scan the honeypot.

The honeypot used for defense for this project is opencanary. Opencanary was chosen because of its familiarity due to classes as well as its simplicity and usability with the rest of the groups implementation. The usability stems from opencanary being built with docker and allows for our group to seamlessly integrate opencanary with our other dockerfiles allowing for the configuration to be quick and, once the configuration file is set building and running the dockerfile easy. The easy deployment of this honeypot will work perfectly for the expected attacks further into the project and will allow deployment of the honeypot on a new server to be easily contained in a script that calls the build and run command.

References

- [1] Miguel Grinberg. *Flask web development*. " O'Reilly Media, Inc.", 2018.
- [2] Past Weather. National weather service. URL: <https://www.weather.gov>, 2009.