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 teammate 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 a URL to physical location feature as well as a URL to the 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. We would like to give extra thanks to Alex Moomaw and Lewis Thomas for helping us and giving advice on our implementation.

2 Related Work

related work section

3 Threat Model

threat modeling (Thomas)

4 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).

5 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. [2]

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

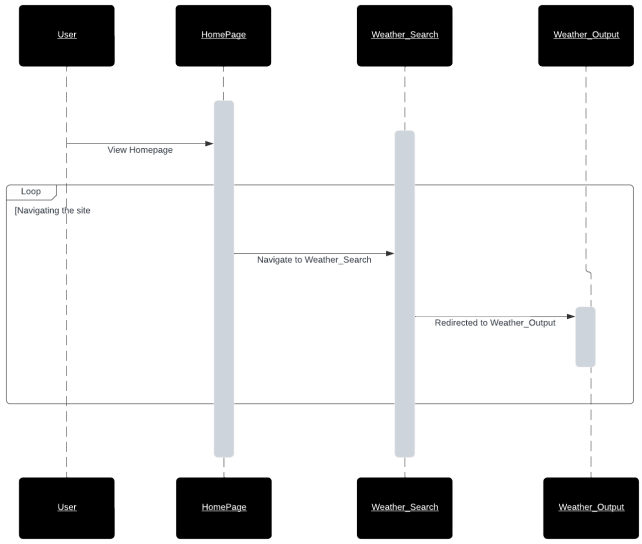

Figure 1: Environment Diagram

6 Services

This web application incorporates two API's in a flask server. One service takes a URL and outputs the address of the registered owner based on the whois [1] registry. The other uses that address and another weather API [**?**, 4] 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.

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 [4] 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.

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.

7 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.

8 Defense

In this section we will discuss the implementation of defense for our group project in detail. Our group's approach to defense includes a two part system, consisting of a honeypot and iptable rules to control traffic. The goal of the honeypot is to provide an avenue for investigation that distracts from the actual critical infrastructure. Once this unauthorized traffic occurs, the rules that we created drops the connection. This slows down or may entirely stop unauthorized users who have the intent to scan the server and try to connect or exploit it. With fake services on the honey pot, their attention will be diverted to multiple red herrings. This defensive approach assumes authorized users will know what is in scope and not try to connect to or scan the honeypot.

The honeypot we chose for this project is opencanary [3]. Opencanary was selected because of our familiarity with it, as well as its simplicity and interoperability with the rest of the group's implementation. Opencanary integrated into the sustem easily because it's built with docker, and allows our group to seamlessly integrate it with our other dockerfiles. This allows for quick configuration, and once the configuration file is set, building and running the dockerfile is quick with simple integegration. The ease of deployment works perfectly for the attacks that we anticipated that were covered in the Threat Model in section 3, and will allow transferring of the honeypot to a new server to be easily handled with a single script.

Traffic rules set by iptables include a very basic approach to discouraging traffic on the machine. Opencanary has enabled a large volume of ports that seem promising to exploit. After an initial scan, the attacker would recognize these and begin to press on these attack surfaces. Unbeknownst to them, all traffic going to these ports will be dropped with no reason or message given to the attacker. Using DROP instead of REJECT provides the client no insight into why their connection is unsuccessful. On their side, it just appears that their request is hanging forever. This simple yet effective measure is a major piece of our defensive strategy. Opencanary's large volume of unnecessary ports in tandem with these rules will cause major disruptions to any penetration attempts, guaranteeing a strong level of security for the system.

9 Evaluation

this is the evaluation section

10 Results

results section

11 Conclusion

conclusion section (Nick)

References

[1] Leslie Daigle. WHOIS Protocol Specification. RFC 3912, September 2004.

[2] Miguel Grinberg. *Flask web development*. " O'Reilly Media, Inc.", 2018.

[3] thnkst. Opencanary. *https://github.com/thinkst/opencanary*, 2024.

[4] Past Weather. National weather service. *URL: https://www. weather. gov*, 2009.