OPC-UA Write Up - Red Alert ICS CTF

Summary

The Red Alert ICS CTF is an annual CTF held by NSHC Security at DEF CON. During DEF CON 31, our team won the CTF, earning a DEF CON Black Badge.

The CTF had a challenge titled OPC-UA. The challenge prompt was as follows:

> A PLC that sends strange signals from the airport has been spotted.
> Analyzing the signals from the PLC seems to give you a clue on how to control the airport.
> Analyze the signals from the PLC to get the 35x35 QR code.
>
> IP : 192.168.50.49 4840

The challenge requires the player to learn about the OPC UA protocol; track, locate and utilize a library in their programming language of choice to interact with this protocol; collect 35 object values within the OPC UA service every second, at least 70 times, to gather enough data to generate the QR code; and track, locate and utilize a library in their programming language of choice to generate a QR code image given this data.

The following was used to solve this challenge:

- ChatGPT: To generate code templates for data collection and QR code image generation
- Python and libraries opcua-asyncio and Pillow: To interact with the PLC and generate the QR code image
- Python project OPC UA Simulation Server: For solver development, as the CTF was open for a limited time over three days
- Python project FreeOpcUa Client: For initial data collection against the OPC UA service.

Details

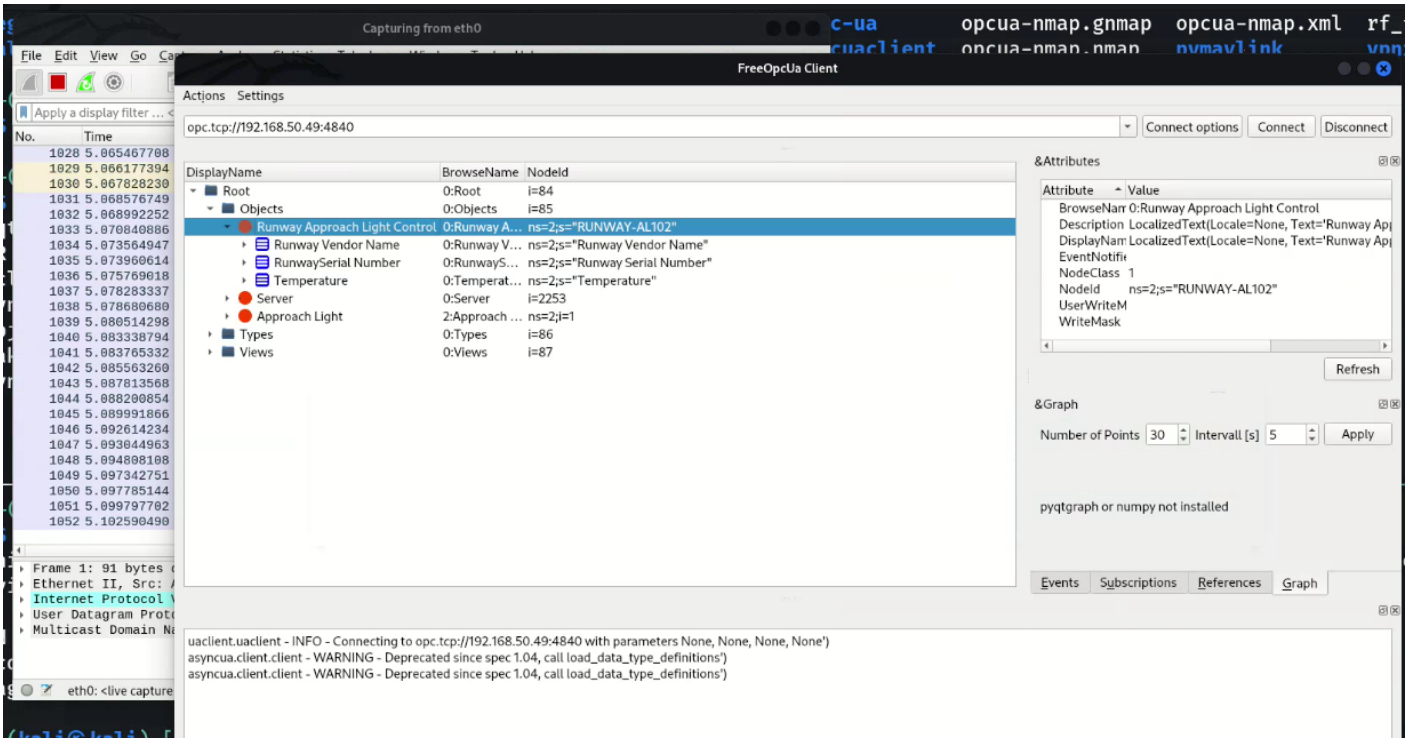
Background

OPC Unified Architecture (OPC UA) is a protocol developed by the Open Platform Communications (OPC) Foundation for use in programmable logic controllers (PLCs) commonly found in industrial control systems. The protocol is open, allowing for wider adoption by PLC manufacturers, and provides newer features such as X.509 client certificate authentication, in-transit encryption, data subscription and method calls.

Walkthrough

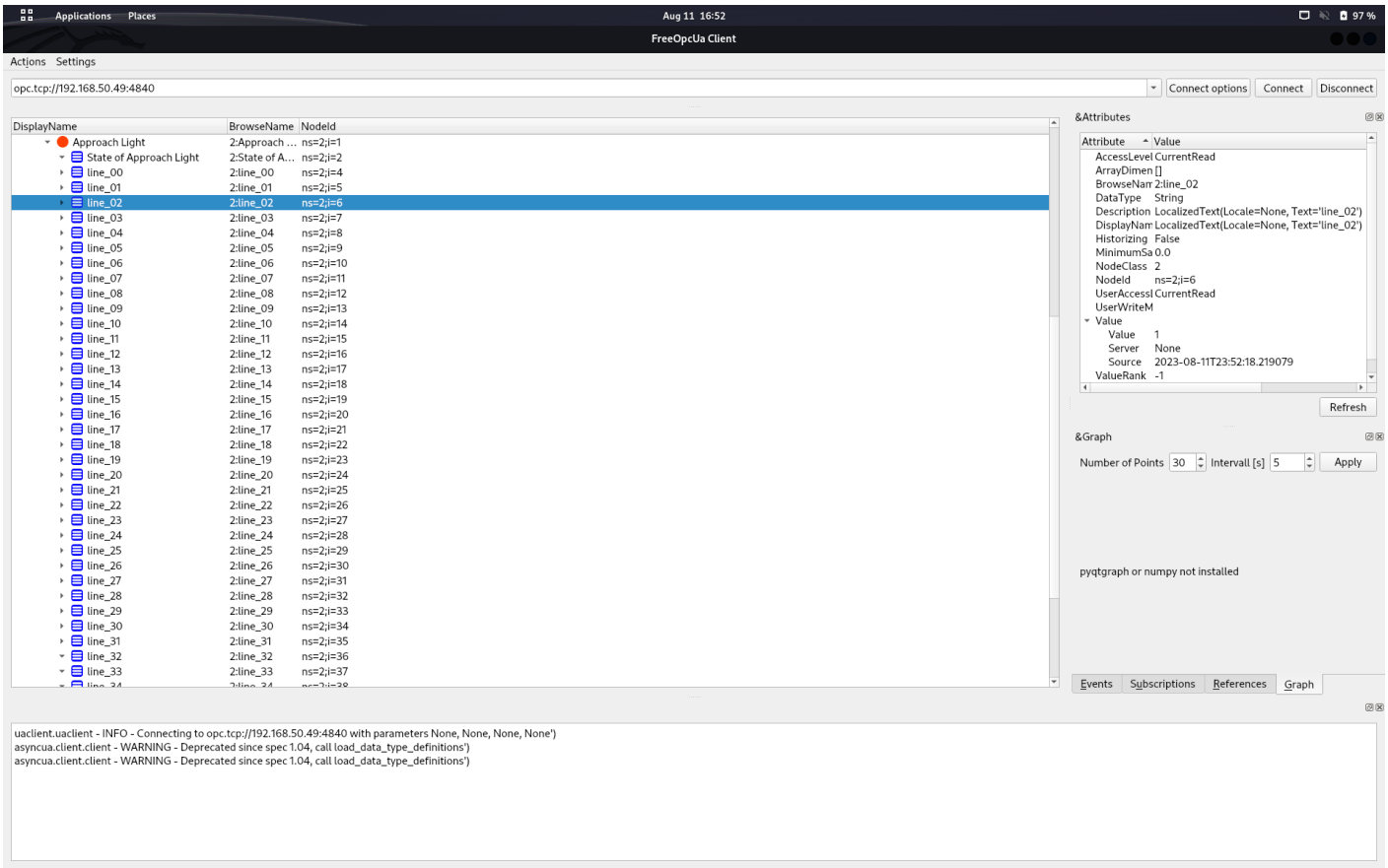
Initial Look

I connected to the service using FreeOpcUa Client (with a packet capture running, of course).



This screen shows us a few things. First, anonymous, unauthenticated sign in is allowed. Second, two custom objects exist: `Runaway Approach Light Control` and `Approach Light`. The `Server` object is standard in PLCs that use the OPC UA protocol.

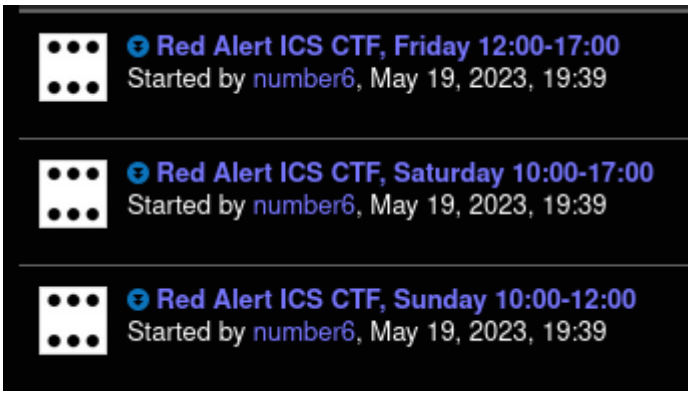
`Runaway Approach Light Control` is a red herring. `Approach Light`, however, looked interesting:



We can see `line_xx` objects, where `xx` is a two-digit number from 0 to 34 (35 in total). Each value is either a 1 or a 0 and changes approximately once a second. There is also a `ts` object, which dumps a timestamp in the ISO 8601 format. `State of Approach Light` turned out to be a red herring.

When initially solving the challenge, the line "analyze the signals from the PLC to get the 35x35 QR code" was missing from the challenge prompt. As a result, I spent many hours attempting to see if there was a flag hidden in a configuration within the `Server` object, or within the defined `Types`. Several hours were spent identifying OPC UA enumeration tools to see if FreeOpcUa Client missed something within the service. As such, I chased rabbit holes all of Friday, and I submitted zero flags. On the second day, the CTF organizers explicitly stated that we needed to derive a 35x35 QR code, and the challenge became far more straightforward.

Simulation, Research and Development



The Red Alert ICS CTF at DEF CON 31 was open for a limited time over three days, per the image above. With this limited time, we almost certainly would be unable to complete enough challenges to beat out the competition and win first place; however, if enough information was collected on the challenges, some of them could be done offline. Our team prioritized challenges that could only be done during competition hours, leaving the rest for the time-to-grind-this-out-and-sleep-at-3AM hours. Ultimately, this strategy was crucial to winning the CTF and the DEF CON 31 Black Badge that year.

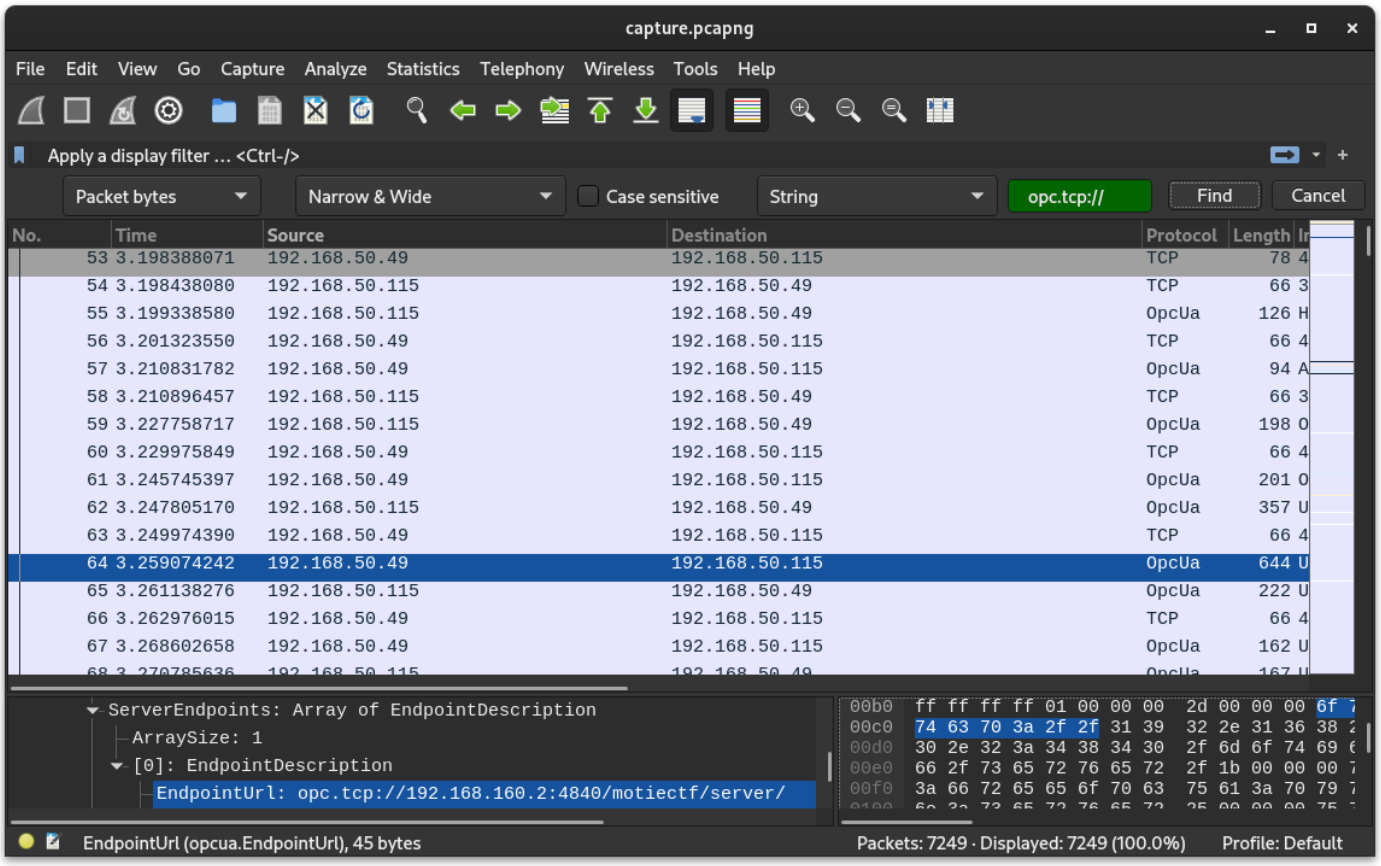
During the competition hours Friday, I primarily focused on other challenges. I did end up finding the opcua-asyncio Python library, which I deemed too complicated to work with, and the OPC UA Simulation Server, which I deemed irrelevant. I still had no idea how to convert seemingly-random 1s and 0s into a QR code, nor how to use the FreeOpcUa Client to effectively collect this data. I put the OPC-UA challenge into the back of my mind, instead targeting other, easier challenges.

That evening, in the middle of dinner, it hit me. The simulation server can be used to recreate the PLC and develop code. The 1's and 0's could be black and white pixels, though I had no idea how to turn that into an image. The timestamp object—maybe a red herring, at the time I was unsure.

I hurried over to Caesar's Forum at approximately 2230H and into the "chill room", where a DJ played the exact type of cyberpunk music needed to quickly put together a botch-job of a script. It sounded like something out of a Mr. Robot hacking scene, except instead of writing a zero-day for FBI standard-issue smartphones, I was busy Frankensteining example code into something that could solve this challenge.

Overnight, I wrote a script that uses the OPC UA subscription service to collect data using the opcua-asyncio library. The library didn't have the best documentation and was not straight-forward to use. In fact, the README code sample contains a syntax error; a more obscure page contains working example code.

The example code came with some limitations. First, it didn't have a programmatic way to determine the full URL to the OPC UA service; it was something that had to hardcoded. I am certain there is a way to determine this dynamically, but I didn't want to read pages upon pages of documentation. Instead, I relied on the packet capture from the initial interaction to determine the URL:



Note that in this packet capture, the IP address within the URL is incorrect. The URI here is what's important: `/motiectf/server/`. I'm still not quite sure why the IP address is different, but if you know feel free to inform me.

As previously mentioned, I used the subscription service to read from a list of objects I needed to collect on, and save any changes detected via the subscription service to a file. I did not know, however, that the subscription service did not provide timely updates, which would cause the final image to come out unreadable, un-QR-able, unscannable, and absolutely useless. Lesson learned: do not rely on the OPC UA subscription service if you need to poll something quicker than once every few seconds.

I learned this lesson Saturday morning, when the the code didn't produce the data I expected. After debugging for a few hours, I gave up and focused on other challenges that entire day. By competition time on Sunday, I just had two hours to figure it out.

Final Stretch

Within the last two hours on Sunday, I relied on ChatGPT to give me some example code to work with. It did quite well, despite my sleep-deprived, nearly-incomprehensible prompts. While I ended up overhauling the code significantly, having a bug-free, executable example to look at that was close to solving the problem was much better than having to read through documentation, especially in a time crunch.

Data Collector Solution

```
import asyncio
import logging
import pickle

from asyncua import Client

url = "opc.tcp://192.168.50.49:4840/motiectf/server/"

# Define the objects we need. This portion of the code could
# likely be less manual, but I did not bother prettifying due
# to the time crunch.
mapper = {
    'ns=2;i=3': 'ts',
    'ns=2;i=4': '0',
    'ns=2;i=5': '1',
    'ns=2;i=6': '2',
    'ns=2;i=7': '3',
    'ns=2;i=8': '4',
    'ns=2;i=9': '5',
    'ns=2;i=10': '6',
    'ns=2;i=11': '7',
    'ns=2;i=12': '8',
    'ns=2;i=13': '9',
    'ns=2;i=14': '10',
    'ns=2;i=15': '11',
    'ns=2;i=16': '12',
    'ns=2;i=17': '13',
    'ns=2;i=18': '14',
    'ns=2;i=19': '15',
    'ns=2;i=20': '16',
    'ns=2;i=21': '17',
    'ns=2;i=22': '18',
    'ns=2;i=23': '19',
    'ns=2;i=24': '20',
    'ns=2;i=25': '21',
    'ns=2;i=26': '22',
    'ns=2;i=27': '23',
    'ns=2;i=28': '24',
    'ns=2;i=29': '25',
    'ns=2;i=30': '26',
```

```
    'ns=2;i=31': '27',
    'ns=2;i=32': '28',
    'ns=2;i=33': '29',
    'ns=2;i=34': '30',
    'ns=2;i=35': '31',
    'ns=2;i=36': '32',
    'ns=2;i=37': '33',
    'ns=2;i=38': '34',
}

# This objects hold the data we ultimately need to save
# and pass onto the QR code generator
save = dict()
_logger = logging.getLogger(__name__)


async def main():
    async with Client(url=url) as client:
        _logger.info("Root node is: %r", client.nodes.root)
        _logger.info("Objects node is: %r", client.nodes.objects)

        # Node objects have methods to read and write node attributes as well as browse or
populate address space
        _logger.info("Children of root are: %r", await client.nodes.root.get_children())

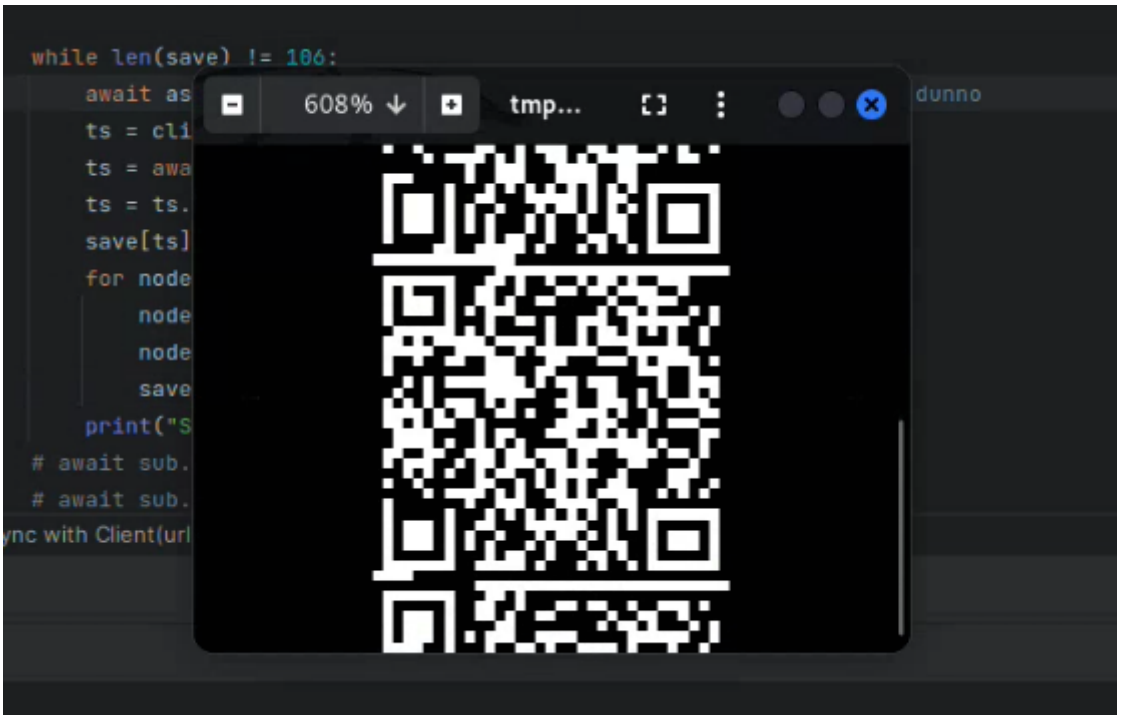
        # Determine the namespace list and grab the very first one
        # since we are required to by the protocol before collecting
        # relevant data
        namespace_list = await client.get_namespace_array()
        namespace = namespace_list[0]
        idx = await client.get_namespace_index(namespace)
        _logger.info("index of our namespace is %s", idx)

        # Generate the necessary node objects to decrease any time delay
        # in object value requests
        nodes = []
        for node in mapper.keys():
            node_obj = client.get_node(node)
            nodes.append(node_obj)
```

```
        while len(save) != 106:
            # Collecting more than one result as there is almost no chance that we will
            # begin collecting data from the very bottom of the QR code, and correcting
            # the collection in post would take longer than just collecting more data than
            # needed.
            await asyncio.sleep(0.1)
            ts = client.get_node("ns=2;i=3")  # Identifier for the ts object
            ts = await ts.read_data_value()
            ts = ts.Value.Value  # Converts value to a Python variable
            if ts not in save.keys():
                # This if statement is CRUCIAL. Without it, you will overwrite
                # collected data, which will almost certainly cause corrupted QR
                # codes
                save[ts] = list()
                for node in nodes:
                    # Collect and save all 35 nodes
                    node_obj = await node.read_data_value()
                    node_obj = node_obj.Value.Value
                    save[ts].append((mapper[str(node)], node_obj))
            print("Saved: " + str(len(save)))
        with open('opcua-log.pkl', 'wb') as f:
            # Save data as a pickle to be able to read it with our QR code generator
            pickle.dump(save, f)


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    asyncio.run(main())
```

The most important portion of the code is line 89, the if statement checking to see if data for a certain timestamp has already been saved and, if it has, to not save it again. The service in the CTF changes data for each object once a second. This means that if we repeatedly save data given a timestamp and overwrite the previous results, we may begin capturing data for one timestamp and end capturing when the timestamp has already changed. This causes the QR code to come out unreadable, like so:



I did not figure out that line until approximately 11 minutes before the competition ended.

QR Code Generator Solution

```
from PIL import Image, ImageDraw
import pickle

width = 35
data = list()

filename = 'opcua-log.pkl'
with open(filename, 'rb') as f:
    save = pickle.load(f)

print(save)  # For debugging

# Determine the image size based on the maximum x-coordinate
image_width = width
image_height = len(save)

# Create a new blank image
image = Image.new("RGB", (image_width, image_height), color="white")
draw = ImageDraw.Draw(image)

# Each timestamp represents the Y axis of the image
```

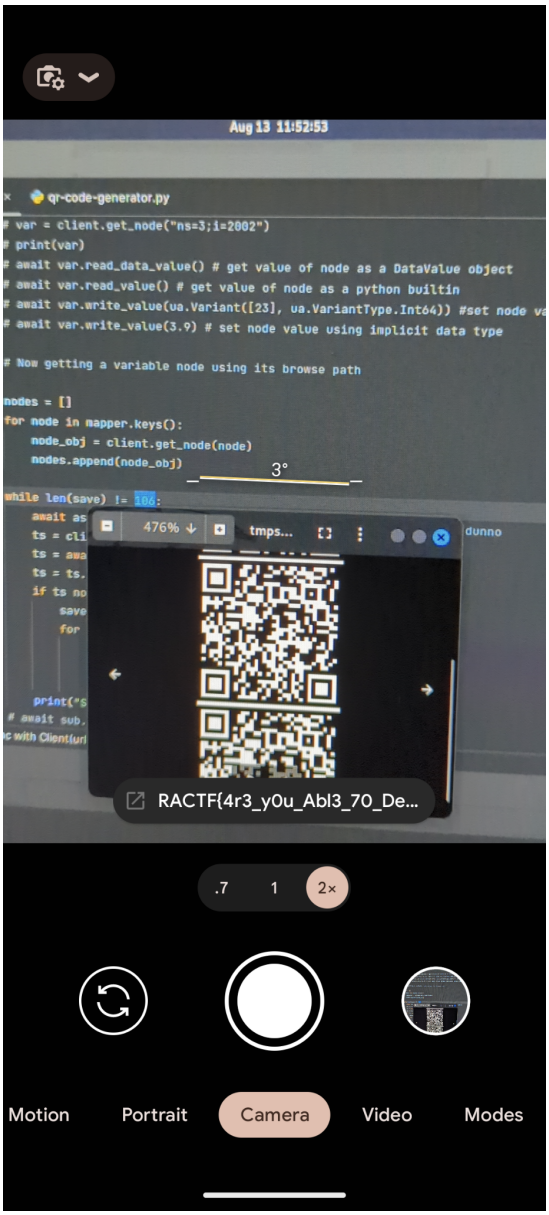
```
for y, ts in enumerate(save.keys()):
    # For each Y axis, if the pixel is 1, then color it black;
    # otherwise, do nothing (leave it white).
    for _, color in enumerate(save[ts]):
        print(color)
        if color[1] == '1':
            draw.point((int(color[0]), y), fill="black")

# Flip the image vertically to match the y-axis orientation
image = image.transpose(Image.FLIP_TOP_BOTTOM)

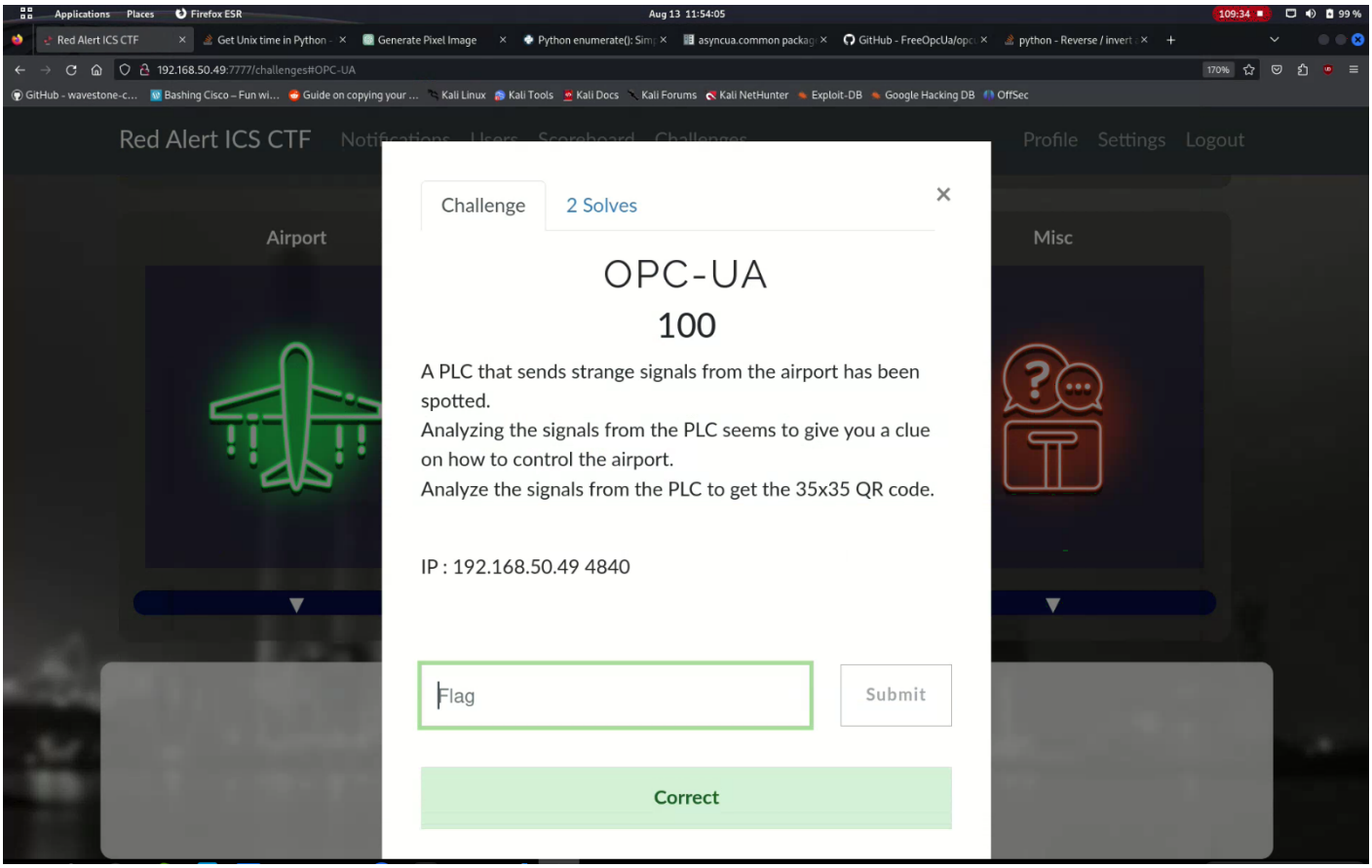
# Display the image
image.show()
```

The QR code generator was fairly straight forward. ChatGPT's example of it was initially irrelevant, as it recommended matplotlib instead of Pillow. Once I asked it to show me some Pillow examples, it came up with something that I could read, understand, and overhaul to make it work with the data format I had.

Final QR Code and Flag



The QR code converted to this flag: `RACTF{4r3_y0u_Abl3_70_DeC0D3_qr}`. Typing that flag from the phone, especially when capital O's and zeroes looked identical, was nervewracking under pressure, but seeing the green "Correct" at 11:54:05, five minutes and fifty-five seconds before the end of the competition, was quite exhilarating.



Aside from our team, ScreamingFist, only one other team solved this challenge during the CTF.

Licensing

This page is licensed under a Creative Commons Attribution 4.0 International License. All code snippets within this page are licensed under a Creative Commons Universal (CC0 1.0) Public Domain Dedication.

Revision #13
Created 2023-08-16 03:15:53 UTC by Henry Reed
Updated 2023-08-18 07:44:13 UTC by Henry Reed