Red Alert ICS CTF - DEF CON 31

- OPC-UA Write Up - Red Alert ICS CTF
- ECDH Write Up - Red Alert ICS CTF

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

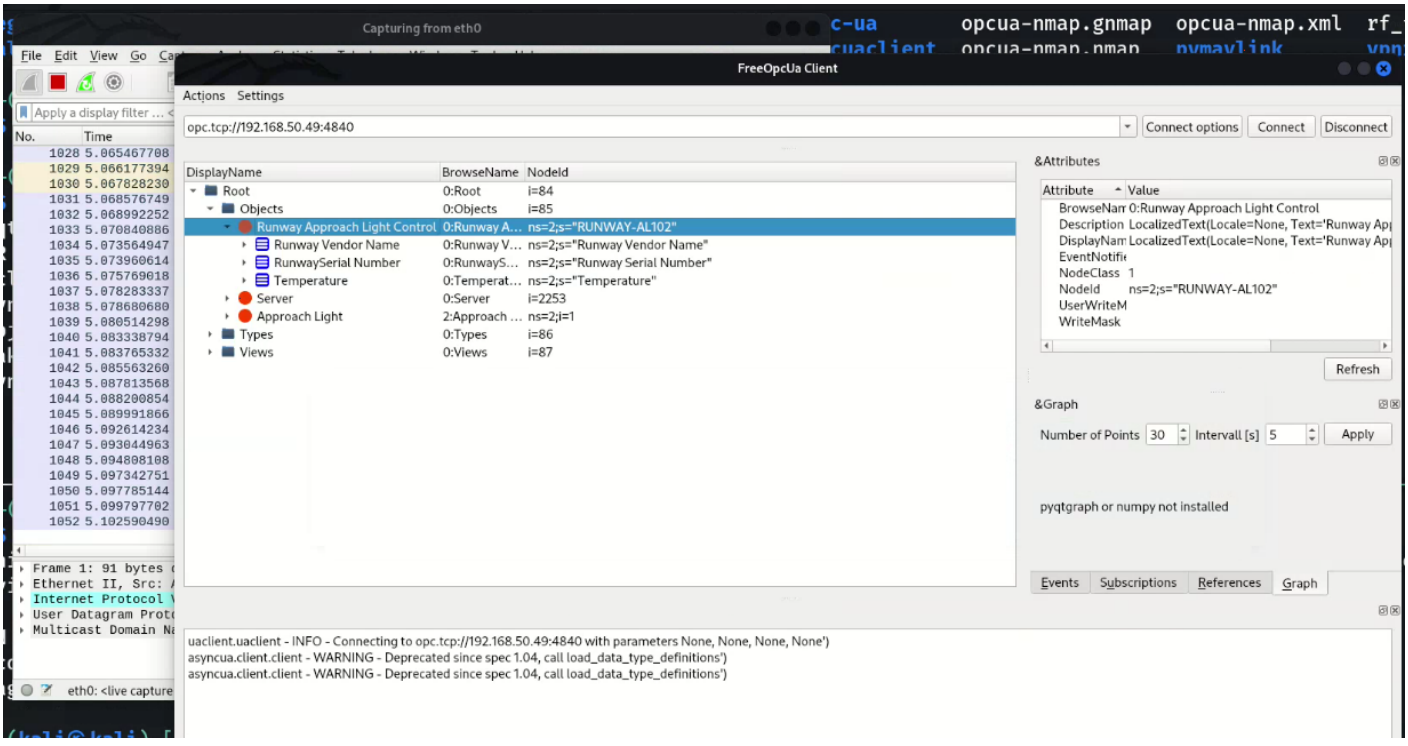
Background

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

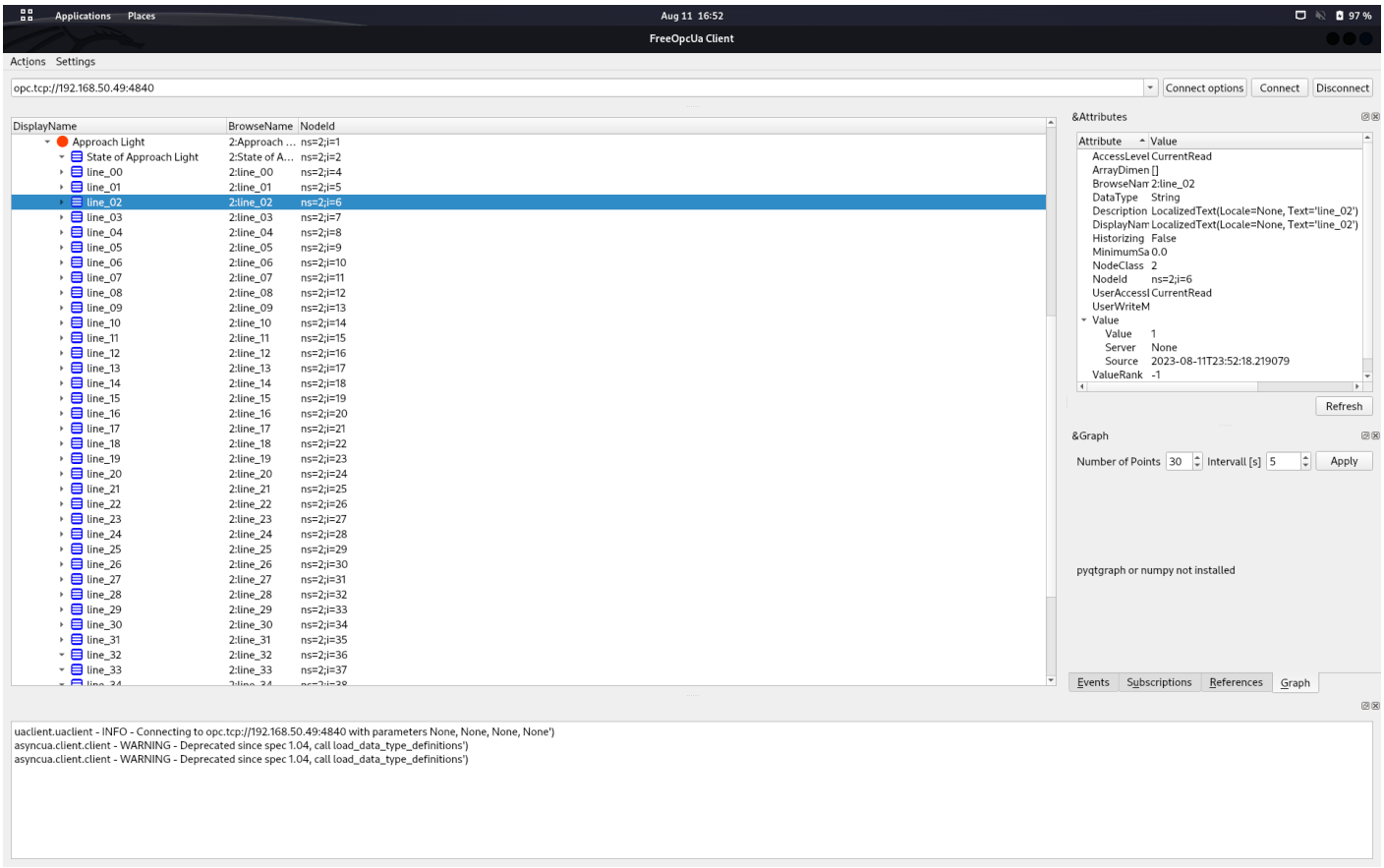
Initial Look

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.

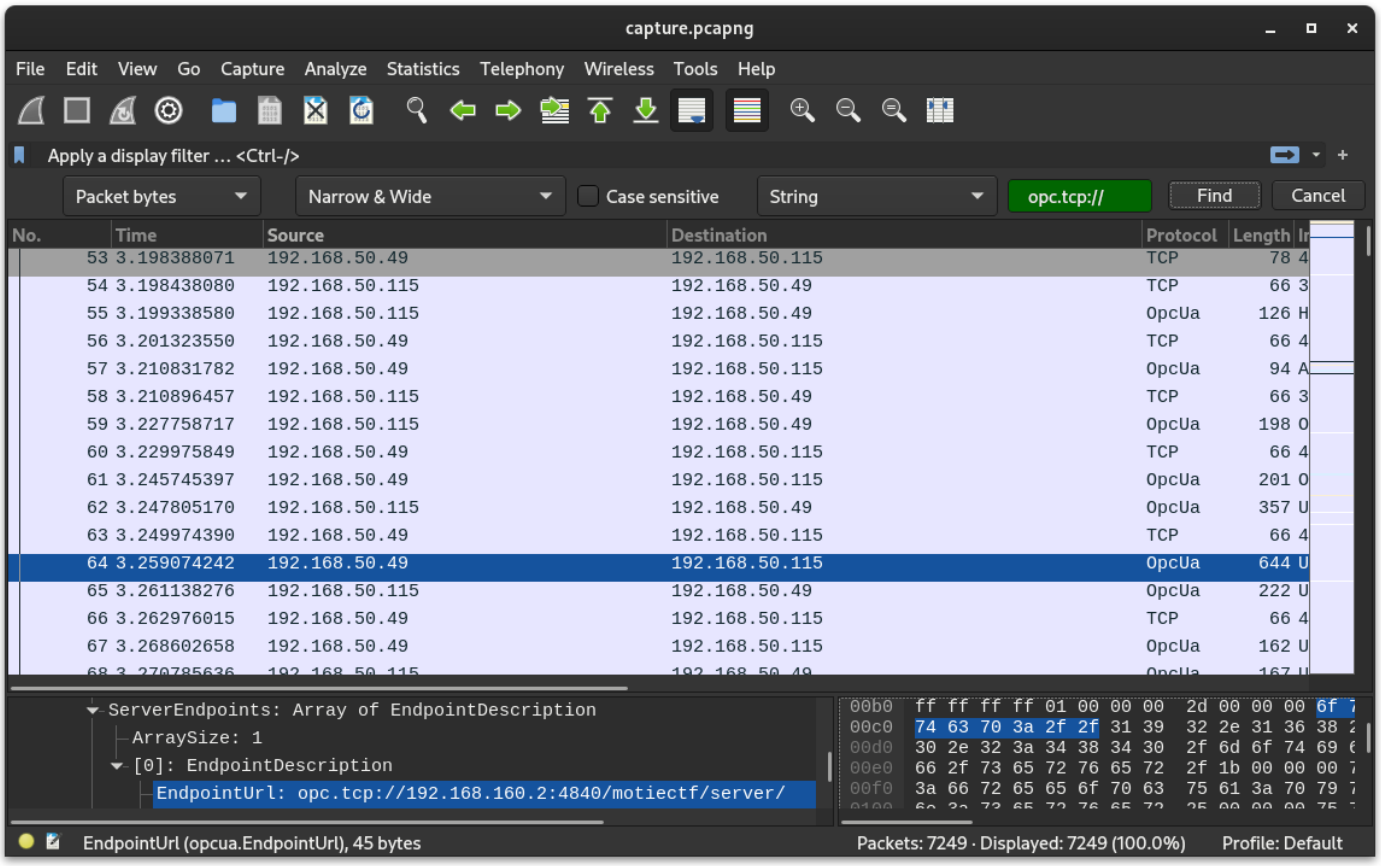
During the competition hours Friday, I primarily focused on other challenges. I did end up finding the opcua-asyncio Python library, which I deemed too complicated to work with, and the OPC UA Simulation Server, which I deemed irrelevant. I still had no idea how to convert seemingly-random 1s and 0s 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.

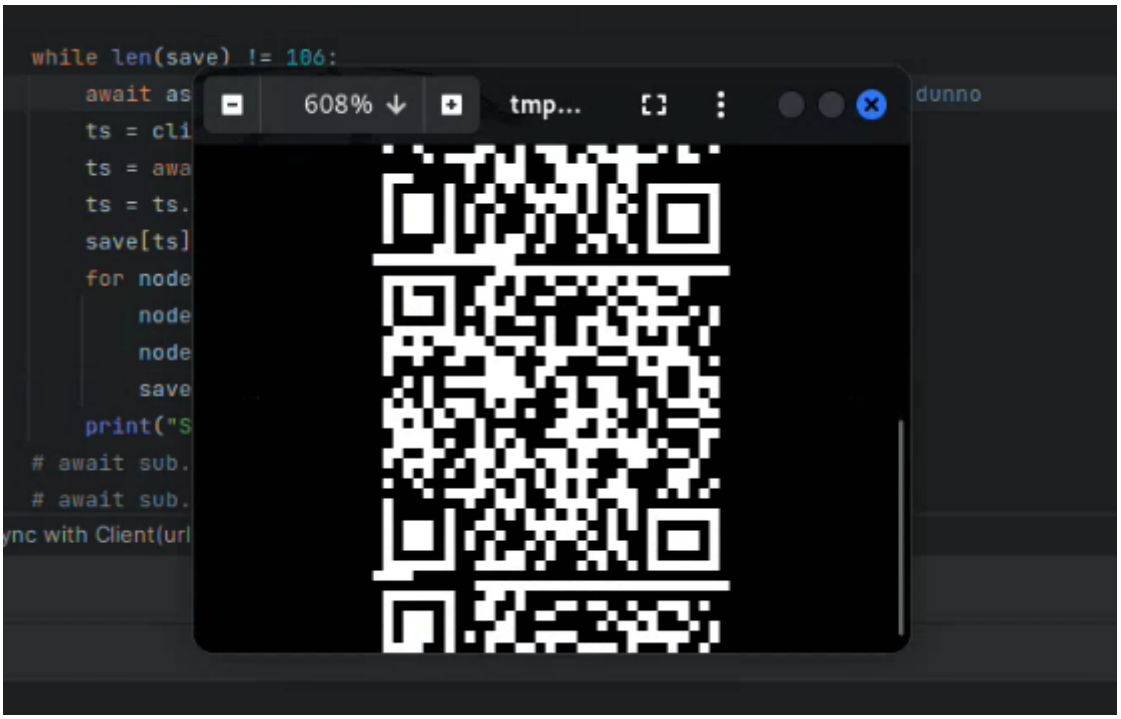
Data Collector Solution

```
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.

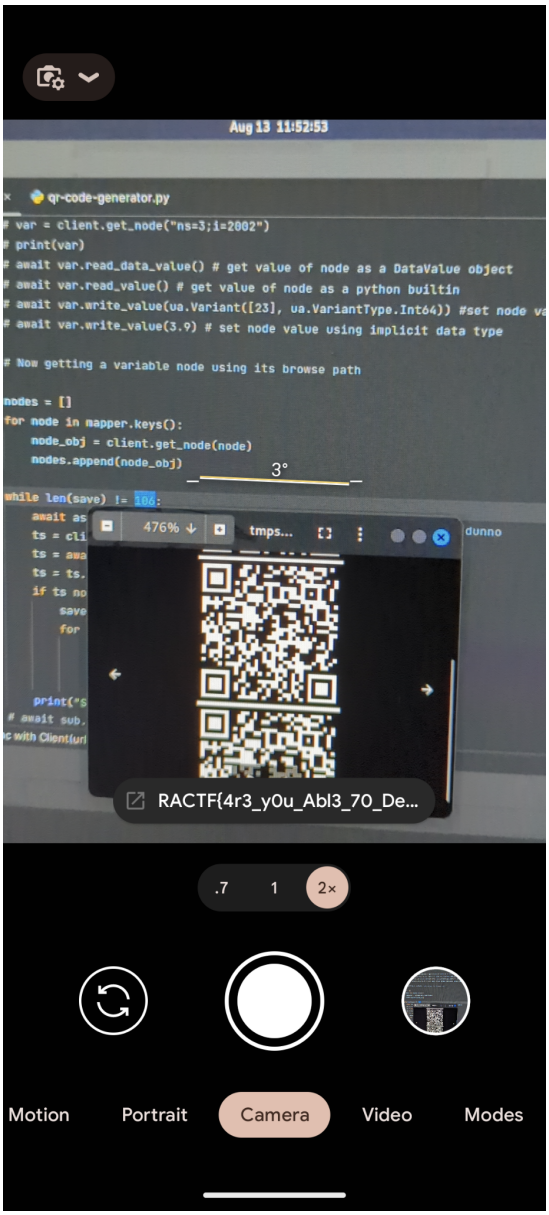
QR Code Generator Solution

```
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
```

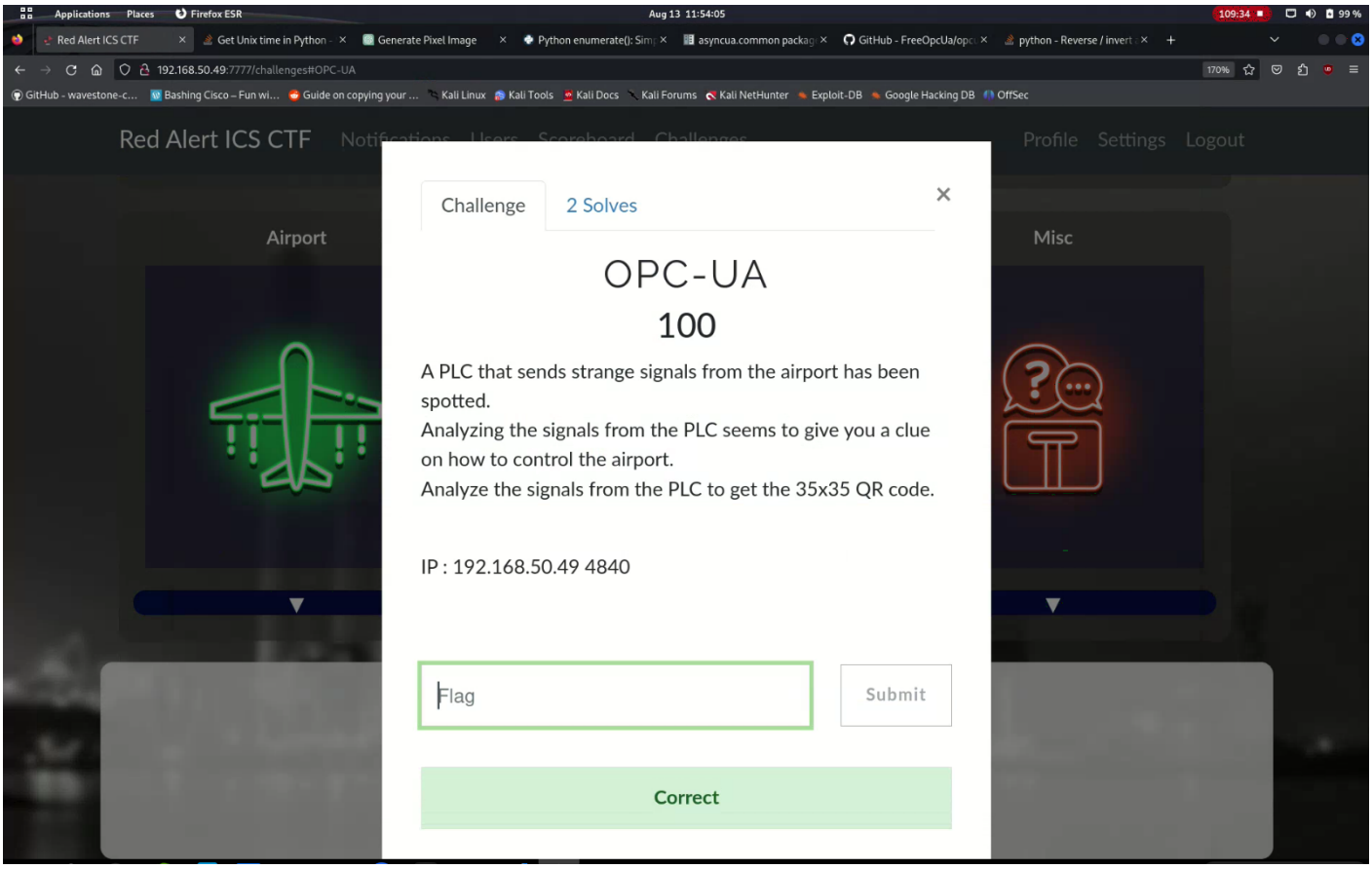
```
for y, ts in enumerate(save.keys()):
    # For each Y axis, if the pixel is 1, then color it black;
    # otherwise, do nothing (leave it white).
    for _, color in enumerate(save[ts]):
        print(color)
        if color[1] == '1':
            draw.point((int(color[0]), y), fill="black")

# 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

ECDH 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 ECDH. The challenge prompt was as follows:

> You have obtained the critical information from the cyber vault for controlling the crane. But the problem is that the file is encrypted. Decrypt the file and capture the flag.

The challenge requires the player to learn about the ECDH protocol and gain familiarity with the PyCryptodome and cryptography Python libraries. The player is given three files (see top left for downloading these files yourself):

- chal.zip.enc: Encrypted zip file containing the flag
- ecdh.py: The code that was used to encrypt the file
- result.txt: Command line output of the encryption process

`ecdh.py` contains a private key while `result.txt` contains a public key. These keys are used to generate a shared secret that is ultimately used with AES to encrypt a file. By knowing the public key of one party and the private key of another, it is trivial to regenerate that same exact shared secret and decrypt the file. The player needs to gain the understanding of ECDH and how it works in theory, then read the documentation for the cryptography library to learn how to import a public key from hex characters and perform the key exchange.

Details

Explaining ecdh.py

From reading the import statements, we can see that AES is imported from PyCryptodome on line 5, and ECDH is imported on line 8.

```
#!/usr/bin/env python3
import hashlib, hmac
import binascii, sys, struct, os
from hexdump import hexdump
from Crypto.Cipher import AES
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, padding, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import PublicFormat
from cryptography.hazmat.primitives.serialization import Encoding
```

The functions `encrypt_file` and `decrypt_file` are overly-complicated AES-CBC encryption and decryption routines, so we can safely ignore those.

We can then see the "define" section of the code, denoted with a comment. This section begins by defining a curve, a signature algorithm, a salt, and an HKDF function:

```
# define ------------------------------------------------------------|
curve = ec.SECP256R1()
signature_algorithm = ec.ECDSA(hashes.SHA256())

salt=b'MOTIECTF-MessageKeys'
hkdf = HKDF(
    algorithm=hashes.SHA256(),
    length=0x30,
    salt=salt,
    info=None,
    backend=default_backend()
)
```

These are all, more or less, irrelevant. We will go over why later. For now, these variables should be left alone.

Next, a private key is generated using the above-defined curve, and its public component is dumped to the screen:

```
ephemeralKey     = ec.generate_private_key(curve, default_backend())   # ephemeral_private
ephemeralPubKey  = ephemeralKey.public_key()
```

```
print("[+] ephemeralPubKey ---")
hexdump(ephemeralPubKey.public_bytes(Encoding.DER, PublicFormat.SubjectPublicKeyInfo))
```

The `ephemeralKey` is *never* printed. Because it is defined from a curve, it will not be possible to regenerate this same key (something something heat death of the universe). The public key, however, is available in the result.txt file:

```
[+] ephemeralPubKey ---
00000000: 30 59 30 13 06 07 2A 86  48 CE 3D 02 01 06 08 2A  0Y0...*.H.=....*
00000010: 86 48 CE 3D 03 01 07 03  42 00 04 1E CF FB A9 9B  .H.=....B.......
00000020: A9 69 9A 73 BA 89 AB 9B  8B 1C 3F 98 9E 77 2A CD  .i.s......?..w*.
00000030: 6D 6A 1B 40 CB 4C 8F 7C  2A 14 43 99 10 DA B0 3F  mj.@.L.|*.C....?
00000040: 0D 87 A9 0D 83 D2 41 11  BF 5A 81 51 85 44 D6 F6  ......A..Z.Q.D..
00000050: 5C FE 54 7F DC 3F E4 E1  A5 66 D8                \.T..?...f.
```

The next three lines are crucial. These sections define a private key; not from a curve, but from raw bytes:

```
privKey     = int.from_bytes(bytes.fromhex("5a55034a6c8ce32e efc745faf7e5e2a8 d24cadd2116ab132
8b634f21f6b21706"), "big")
privKey     = ec.derive_private_key(privKey, curve, default_backend())
pubKey      = privKey.public_key()
```

Next, the shared secret is created, and the AES key and IV are derived from that shared secret using the above-defined HKDF algorithm:

```
shared_key  = ephemeralKey.exchange(ec.ECDH(), pubKey)
derived_key = hkdf.derive(shared_key)
key         = derived_key[:0x20]
iv          = derived_key[0x20:]
```

What does this mean?

The code generates a private key called `ephemeralKey` that is eventually garbage-collected and lost. It also uses another hard-coded private key to generate a public key. That public key is then used with `ephemeralKey` to create what eventually becomes the AES key and IV.

The file is encrypted with AES-CBC using the variable `derived_key` as its key and IV. We want to generate the same `derived_key`. We know that this variable is created via `hkdf.derive(shared_key)`. We don't care what HKDF is or what it does. It only takes in one variable, so it stands to reason if we can generate that variable, we can generate `derived_key` via the same exact function call. HKDF takes in `shared_key`, so our ultimate goal is to create the same `shared_key` variable and call the

`decrypt_file` function.

Let's break down the call that defines `shared_key`. The function is `ephemeralKey.exchange`. `ephemeralKey` is created using `ec.generate_private_key`. `ec` is imported from `cryptography.hazmat.primitives.asymmetric`. Time to read some documentation for that import, available here:

Elliptic curve cryptography

cryptography.hazmat.primitives.asymmetric.ec.generate_private_key(*curve*) [source]

New in version 0.5.

Generate a new private key on `curve`.

Parameters: **curve** – An instance of `EllipticCurve`.
Returns: A new instance of `EllipticCurvePrivateKey`.

We see that the return of `ec.generate_private_key` is a "new instance of `EllipticCurvePrivateKey`." Clicking on that, we see at the very top the same `exchange` function we saw before in `shared_key = ephemeralKey.exchange`:

class **cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey** [source]

New in version 0.5.

An elliptic curve private key for use with an algorithm such as ECDSA.

exchange(*algorithm*, *peer_public_key*) [source]

New in version 1.1.

Performs a key exchange operation using the provided algorithm with the peer's public key.

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

Parameters:
- **algorithm** – The key exchange algorithm, currently only `ECDH` is supported.
- **peer_public_key** (*EllipticCurvePublicKey*) – The public key for the peer.

Returns bytes: A shared key.

The two variables are a public key and any key exchange algorithm, as long as it's ECDH. For some reason, this makes me think of cars and their varied colors, not sure why though. Anyway, we know

that the function call uses ECDH, as the full line is `shared_key = ephemeralKey.exchange(ec.ECDH(), pubKey)`. Clicking on ECDH, we get a big wall of text and code. We're close.

class **cryptography.hazmat.primitives.asymmetric.ec.ECDH** [source]

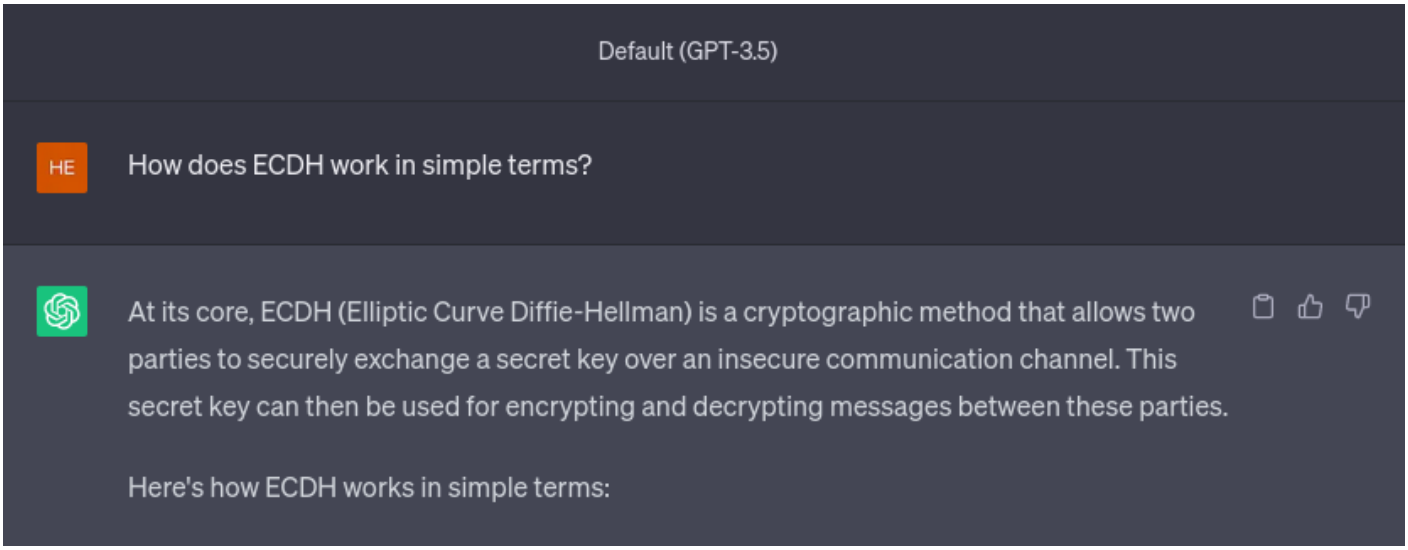
New in version 1.1.

The Elliptic Curve Diffie-Hellman Key Exchange algorithm standardized in NIST publication 800-56A.

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

Note that while elliptic curve keys can be used for both signing and key exchange, this is bad cryptographic practice. Instead, users should generate separate signing and ECDH keys.

We now know that the AES-CBC key is created using "the Elliptic Curve Diffie-Hellman Key Exchange algorithm standardized in NIST publication 800-56A." Now, as much as I love NIST, I am not reading 152 pages during a CTF. Similarly, the Wikipedia page looks equally as scary as the NIST paper—there are far too many weird-looking Greek letters. I need someone to explain to me what ECDH does, do so using layman terms, and answer any of my follow up questions quickly, without any snark, without asking me what I'm trying to do, without telling me to do something different, without asking me to take my comments to Stack Overflow chat, and with the knowledge that can only be created by scraping the entire internet.



Oh, OpenAI, how I love you. Our friend, ChatGPT, gives us a clear answer. The most important part is the following:

3. **Key Exchange**: When both parties have their own private and public keys, they exchange their public keys over the insecure channel.
4. **Shared Secret Calculation**: Using the received public key from the other party and their own private key, each party performs another mathematical operation. This operation results in a shared secret that is the same for both parties, even though they have different private keys.

We have the hard-coded private key and the ephemeral public key. All we need to do is call the same functions but with different keys, as the "operation results in a shared secret that is the same for both parties, even though they have different private keys."

Solution:

We just need to make a few adjustments. We replace the `ephemeralPubKey` with the public key from `result.txt`:

```
ephemeralPubKey  = serialization.load_der_public_key(bytes.fromhex("3059301306072A86 48CE3D020106082A 8648CE3D03010703 4200041ECFFBA99B A9699A73BA89AB9B 8B1C3F989E772ACD 6D6A1B40CB4C8F7C 2A14439910DAB03F 0D87A90D83D24111 BF5A81518544D6F6 5CFE547FDC3FE4E1 A566D8"))
```

Instead of creating the shared_key using the ephemeralKey and pubKey, we instead use privKey and ephemeralPubKey:

```
shared_key  = privKey.exchange(ec.ECDH(), ephemeralPubKey)
```

We update the file name:

```
filename    = 'chal.zip.enc'
```

And we decrypt, instead of encrypting:

```
decrypt_file(key, iv, filename)
```

Unzipping the file we get the flag `RACTF{Elliptic_curv3_Diffi3_H3llm4n!!}`.

Full Solution:

```
#!/usr/bin/env python3
import hashlib, hmac
import binascii, sys, struct, os
```

```
from hexdump import hexdump
from Crypto.Cipher import AES
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, padding, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import PublicFormat
from cryptography.hazmat.primitives.serialization import Encoding

def encrypt_file(key, iv, in_filename, out_filename=None, chunksize=64*1024):
    if not out_filename:
        out_filename = in_filename + '.enc'

    encryptor = AES.new(key, AES.MODE_CBC, iv)
    filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            outfile.write(struct.pack('<Q', filesize))

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += b' ' * (16 - len(chunk) % 16)

                outfile.write(encryptor.encrypt(chunk))

def decrypt_file(key, iv, in_filename, out_filename=None, chunksize=24*1024):
    if not out_filename:
        out_filename = os.path.splitext(in_filename)[0]

    with open(in_filename, 'rb') as infile:
        origsize = struct.unpack('<Q', infile.read(struct.calcsize('Q')))[0]
        decryptor = AES.new(key, AES.MODE_CBC, iv)

        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
```

```
            if len(chunk) == 0:
                break
            outfile.write(decryptor.decrypt(chunk))

        outfile.truncate(origsize)

# define ----------------------------------------------------------------|
curve = ec.SECP256R1()
signature_algorithm = ec.ECDSA(hashes.SHA256())

salt=b'MOTIECTF-MessageKeys'
hkdf = HKDF(
    algorithm=hashes.SHA256(),
    length=0x30,
    salt=salt,
    info=None,
    backend=default_backend()
)

#ephemeralKey     = ec.generate_private_key(curve, default_backend())   # ephemeral_private
ephemeralPubKey  = serialization.load_der_public_key(bytes.fromhex("3059301306072A86 48CE3D020106082A
8648CE3D03010703 4200041ECFFBA99B A9699A73BA89AB9B 8B1C3F989E772ACD 6D6A1B40CB4C8F7C
2A14439910DAB03F 0D87A90D83D24111 BF5A81518544D6F6 5CFE547FDC3FE4E1 A566D8"))
print("[+] ephemeralPubKey ---")
hexdump(ephemeralPubKey.public_bytes(Encoding.DER, PublicFormat.SubjectPublicKeyInfo))

privKey     = int.from_bytes(bytes.fromhex("5a55034a6c8ce32e efc745faf7e5e2a8 d24cadd2116ab132
8b634f21f6b21706"), "big")
privKey     = ec.derive_private_key(privKey, curve, default_backend())
pubKey      = privKey.public_key()

shared_key  = privKey.exchange(ec.ECDH(), ephemeralPubKey)
derived_key = hkdf.derive(shared_key)
key         = derived_key[:0x20]
iv          = derived_key[0x20:]

# Encrypting file
print("[+] Decrypting file")
filename    = 'chal.zip.enc'
decrypt_file(key, iv, filename)
```

```
print("[+] Decryption DONE")
```

Licensing