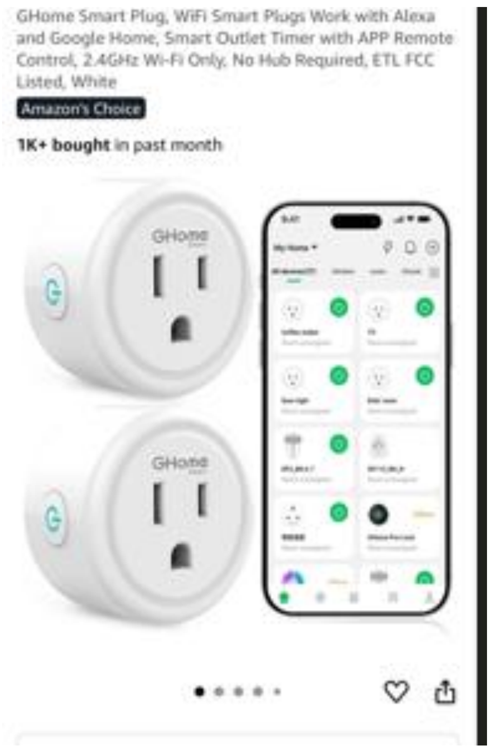
GHome Smart Plug Initial Analysis – Gabriel Adams

A few weeks back I decided to purchase a new IoT device target with the intention of publishing my findings. I think this past year I have really seen the benefit of making the work that I do public for others to see; it works as both a proof of skill for employment as well as allowing for larger cooperation with other immensely smart people to give their thoughts in places where I might have missed.

This target was chosen after I asked ChatGPT: "Find me an IoT device that is cheap, can be purchased on Amazon, has an app, and is made by a Chinese company." Funny enough, it did a great job and found me a few great candidates but I picked this one.


GHome Smart Plug, WiFi Smart Plugs Work with Alexa and Google Home, Smart Outlet Timer with APP Remote Control, 2.4GHz Wi-Fi Only, No Hub Required, ETL FCC Listed, White

Amazon's Choice

1K+ bought in past month

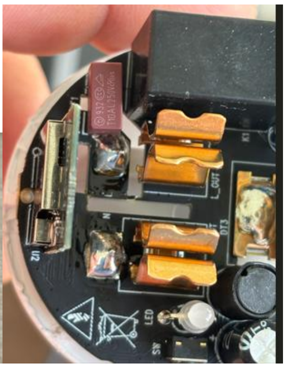
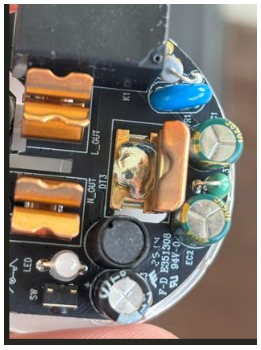
So I ordered two of them and a few days later had the box with the two come in. From here on out this writeup will follow these steps that I took in my analysis:

1. **Breakdown**
2. **Identifying Pins**
3. **Finding the chip type**
4. **Finding the software**
5. **Grounding the CEN pin to bypass ROM check**
6. **Separating out the firmware memory layout.**
7. **Investigating the app and bootloader.**
8. **Different Wireshark Captures**
9. **Deciding to look at the app**
10. **Setting up the proxy.**
11. **Looking at traffic in MitmWeb using MitmProxy.**
12. **Closing remarks and future work.**

BreakDown

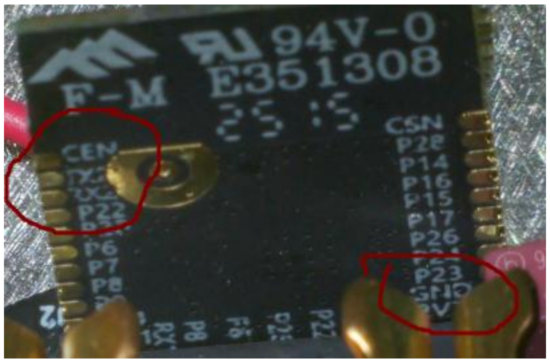
The device came in looking like the following pictures:

And then the following images are during different steps of the process of taking it apart to view its makeup.

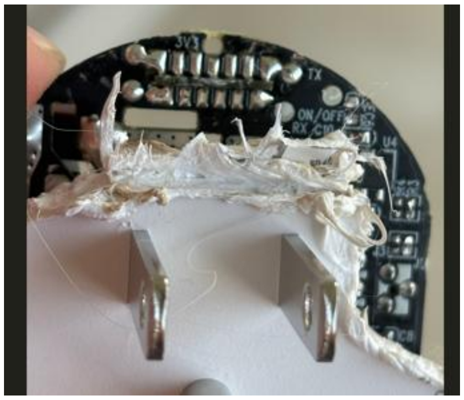



Identifying Pins

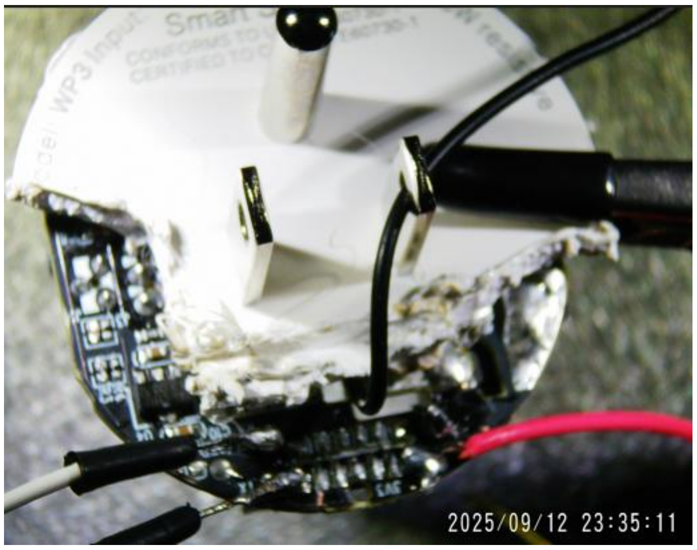
One of the exposed chips was the MCU that displayed GND, TX, RX, and 3V3 pins. This tipped me off to UART. Also remember that CEN pinout.



I didn't have the equipment yet to use these exposed pinouts to get a connection to the device (I have since got me some PCBites), but then I saw there were debug interface pads on the back, covered by the casing. I made the *not so smart* decision to use my soldering iron like one of those "cut it like butter with this 3000-degree knife" videos.



So I got out my multimeter and figured out the tracing to these pads to then solder them on.



Finding the chip type

Before attempting to get a UART shell, I wanted to investigate the chip on board. I found that the GHome brand is owned by the company called 'Tuya' which get's confirmed later. The following was the report I made to track my progress.

The device was branded as *GHome*, but GHome devices are actually re-brands of Tuya smart plugs and switches. Tuya is a major IoT platform provider that sells white-label devices to many vendors. Documentation and teardown communities consistently point out that Tuya hardware is built on **Beken BK72xx Wi-Fi/BT SoCs**.

Later, public firmware builds and strings inside the binary itself confirmed tuyaos-iot with bk7231n. This strongly suggested a

Finding the right software:

BK7231N SoC, which is one of Beken's common MCU's with Wi-Fi and Bluetooth built in that is used in Tuya devices.

- Community reference: [OpenBK7231N project – GitHub]

Once the chip was identified as likely BK72xx, I used **ltchiptool**, an open-source flashing utility that explicitly supports Tuya/Beken chips. It leverages the UART-based boot ROM downloader to read and write firmware.

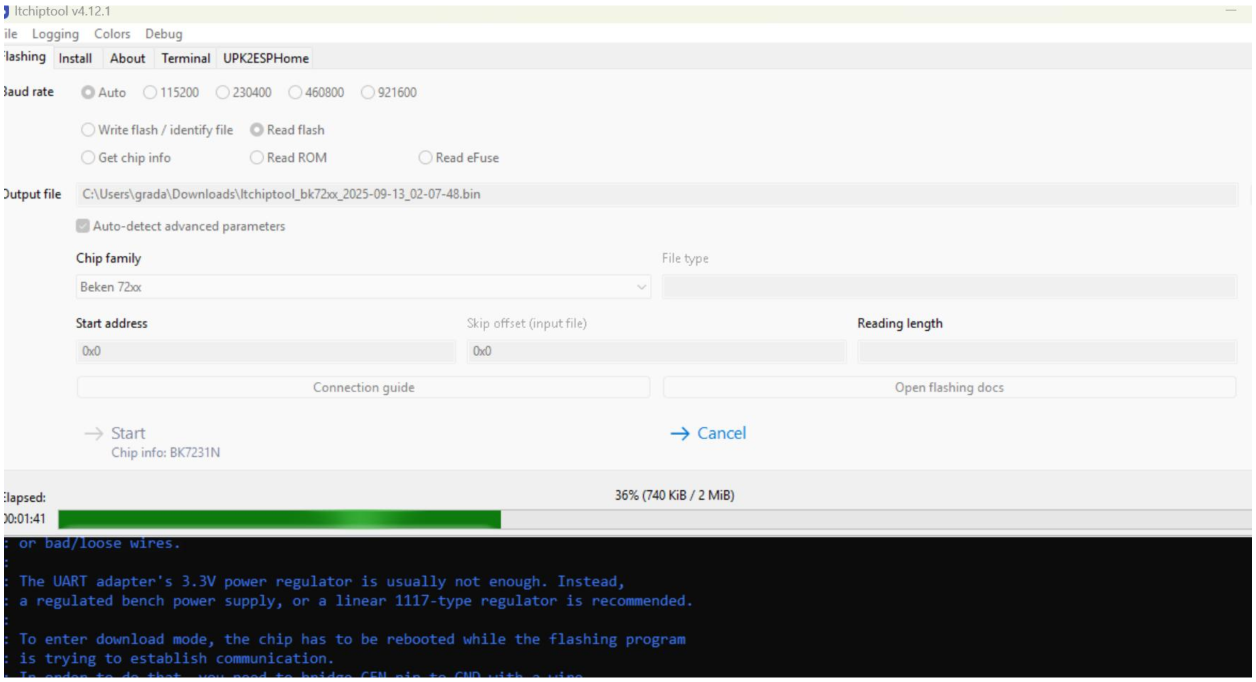
Grounding the CEN pin to bypass ROM check:

By default, the chip boots from internal flash. To dump the firmware, it must be forced into the **boot ROM downloader**. The BK72xx has a built-in UART bootloader that activates if the **CEN (chip enable / reset)** pin is held low during reset/power-up. Shorting CEN to ground tells the chip to skip executing flash firmware and instead wait for serial commands over UART.

Reference: LibreTiny BK72xx flashing guide

Explanation: This mechanism is common in embedded MCUs/SoCs — by asserting reset low (CEN→GND) at the right time, you inhibit normal boot and fall back to ROM code, which is immutable and intended for factory flashing/debugging. If I'm being honest I wanted to say that I glitched the device, but as someone who has tried to pull off a difficult glitch before: this was nowhere close to that. It was really just me standing at a weird angle to hold the pin to the small CEN pad and then when the light turned on from power quickly touching ground with the other end of the wire. The whole time with my tongue out and extreme focus. After a few times the firmware was dumped. The link I used to download the software is here:

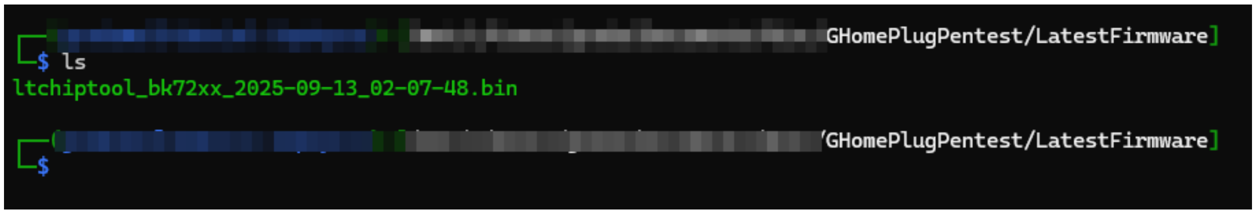
Releases · libretiny-eu/ltchiptool



```
Baud rate   [✓ Auto]  115200  230400  460800  921600

Write flash / identify file   [✓ Read flash]
Get chip info   Read ROM   Read eFuse

Output file   C:\Users\grada\Downloads\ltchiptool_bk72xx_2025-09-13_02-07-48.bin
[✓ Auto-detect advanced parameters]
Chip family   Beken 72xx
File type
Start address   0x0
Skip offset (input file)   0x0
Reading length

Chip info: BK7231N

Elapsed:
00:01:41
36% (740 KiB / 2 MiB)
```

```
: or bad/loose wires.
:
: The UART adapter's 3.3V power regulator is usually not enough. Instead,
: a regulated bench power supply, or a linear 1117-type regulator is recommended.
:
: To enter download mode, the chip has to be rebooted while the flashing program
: is trying to establish communication.
```

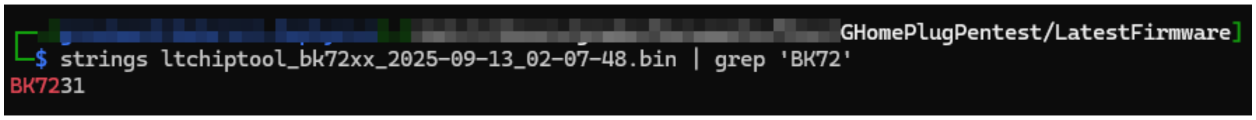
This was after a successful attempt of booting into ROM.



```
┌─[redacted]GHomePlugPentest/LatestFirmware]
└─$ ls
ltchiptool_bk72xx_2025-09-13_02-07-48.bin

┌─[redacted]/GHomePlugPentest/LatestFirmware]
└─$
```

Separating out the firmware memory layout:

We are left with this .bin file. The tool downloaded also has some tools to separate out the different partitions IF we know what the exact chip is.



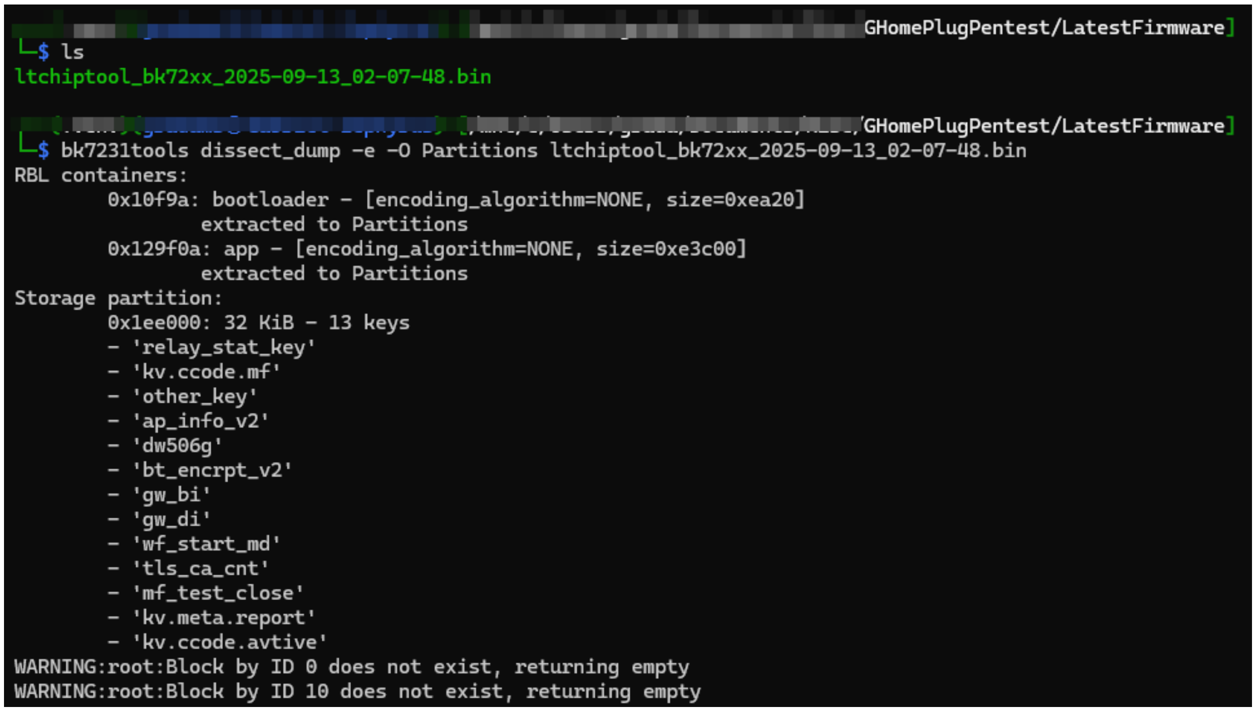
```
┌──[redacted]GHomePlugPentest/LatestFirmware]
└─$ strings ltchiptool_bk72xx_2025-09-13_02-07-48.bin | grep 'BK72'
BK7231
```

That was convenient.

Now admittedly I wanted to use my kali environment to analyze the firmware, so I installed ‘ltchiptool’ through pip:

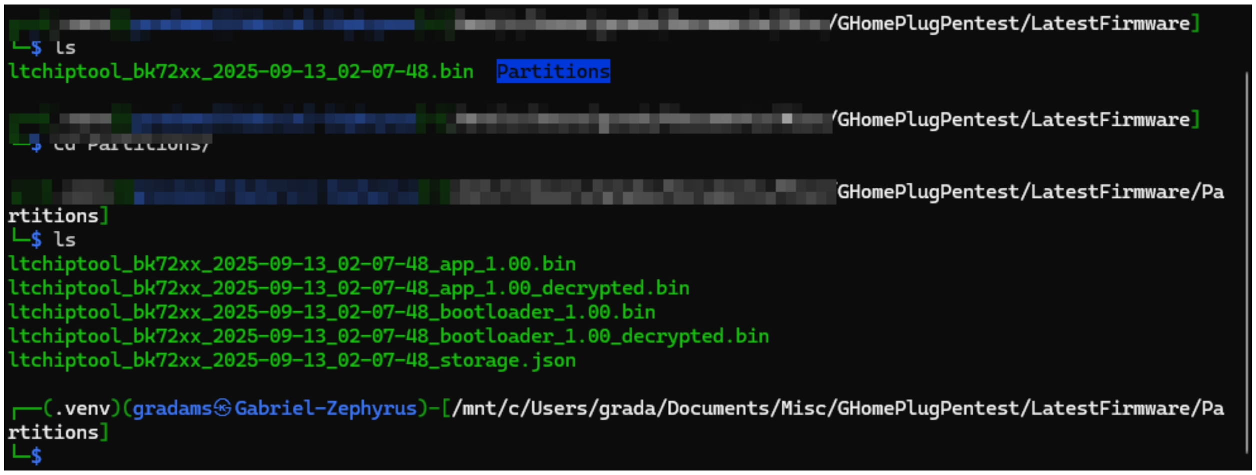
pip install -U ltchiptool

Then, after looking through some documentation (asking chat), I went through the following steps:



```
[redacted]GHomePlugPentest/LatestFirmware]
└─$ ls
ltchiptool_bk72xx_2025-09-13_02-07-48.bin

[redacted]GHomePlugPentest/LatestFirmware]
└─$ bk7231tools dissect_dump -e -O Partitions ltchiptool_bk72xx_2025-09-13_02-07-48.bin
RBL containers:
        0x10f9a: bootloader - [encoding_algorithm=NONE, size=0xea20]
                extracted to Partitions
        0x129f0a: app - [encoding_algorithm=NONE, size=0xe3c00]
                extracted to Partitions
Storage partition:
        0x1ee000: 32 KiB - 13 keys
        - 'relay_stat_key'
        - 'kv.ccode.mf'
        - 'other_key'
        - 'ap_info_v2'
        - 'dw506g'
        - 'bt_encrpt_v2'
        - 'gw_bi'
        - 'gw_di'
        - 'wf_start_md'
        - 'tls_ca_cnt'
        - 'mf_test_close'
        - 'kv.meta.report'
        - 'kv.ccode.avtive'
WARNING:root:Block by ID 0 does not exist, returning empty
WARNING:root:Block by ID 10 does not exist, returning empty
```

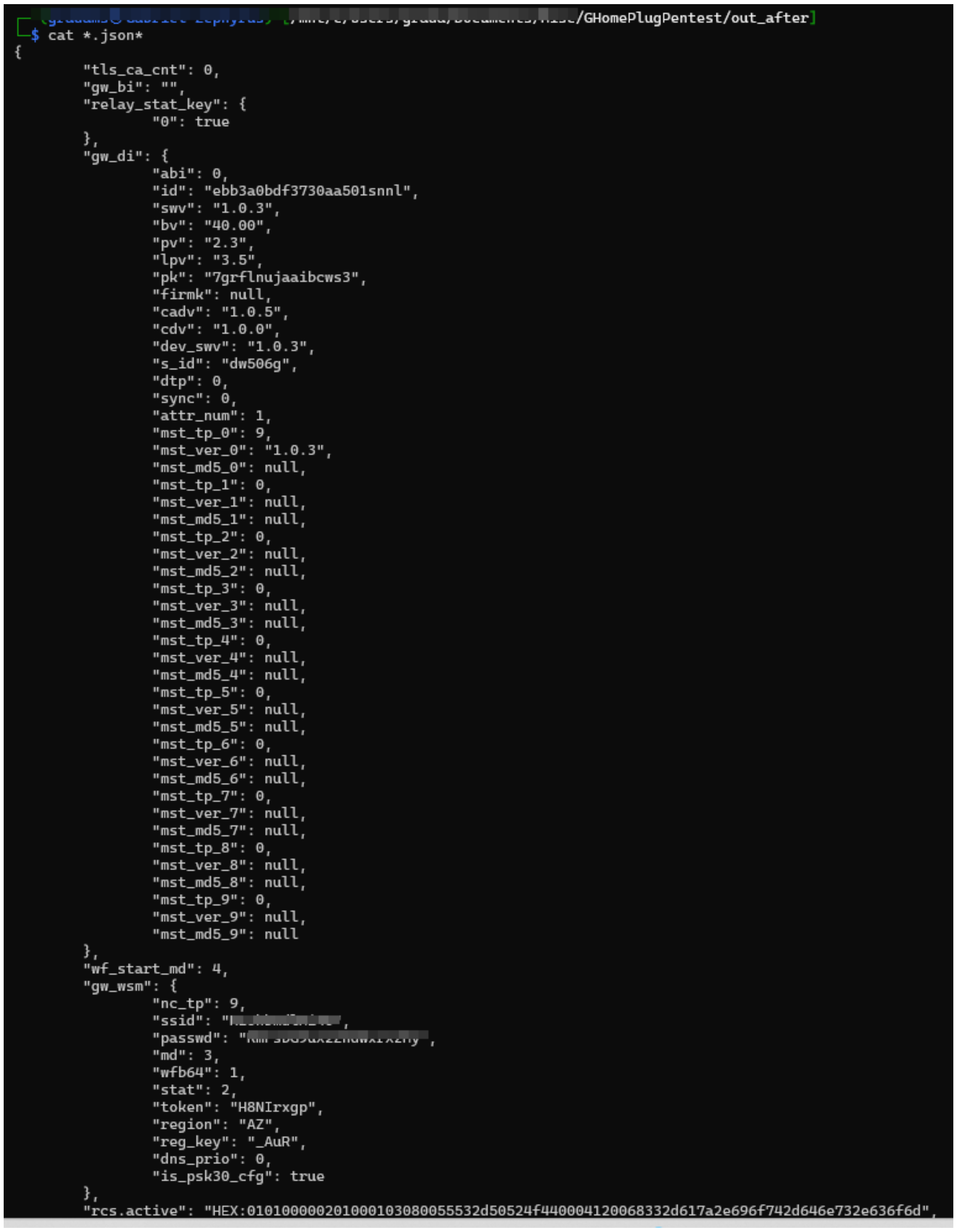
This separates out the different parts of the firmware which leaves us with this:



```
[redacted]/GHomePlugPentest/LatestFirmware]
└─$ ls
ltchiptool_bk72xx_2025-09-13_02-07-48.bin  Partitions

[redacted]/GHomePlugPentest/LatestFirmware]
└─$ cd Partitions/

[redacted]GHomePlugPentest/LatestFirmware/Pa
rtitions]
└─$ ls
ltchiptool_bk72xx_2025-09-13_02-07-48_app_1.00.bin
ltchiptool_bk72xx_2025-09-13_02-07-48_app_1.00_decrypted.bin
ltchiptool_bk72xx_2025-09-13_02-07-48_bootloader_1.00.bin
ltchiptool_bk72xx_2025-09-13_02-07-48_bootloader_1.00_decrypted.bin
ltchiptool_bk72xx_2025-09-13_02-07-48_storage.json

┌──(.venv)(gradams㉿Gabriel-Zephyrus)-[/mnt/c/Users/grada/Documents/Misc/GHomePlugPentest/LatestFirmware/Pa
rtitions]
└─$
```

Investigating the app and bootloader:

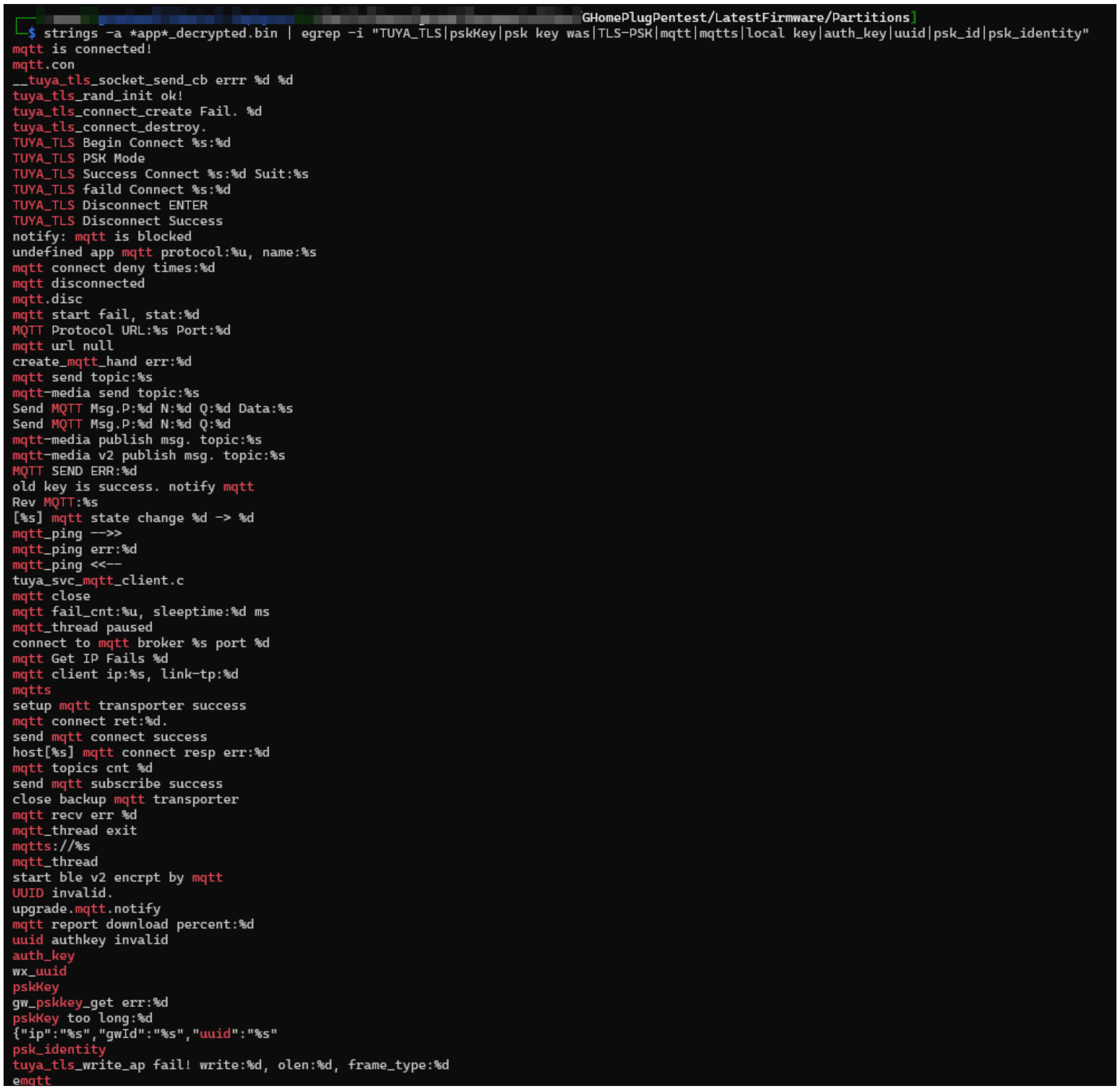
That .json file has some interesting details after the device is completely set up. Once I connected it to my WiFi network, I dumped the firmware again, and it contained the following:



```
┌─[redacted]/GHomePlugPentest/out_after]
└─$ cat *.json*
{
        "tls_ca_cnt": 0,
        "gw_bi": "",
        "relay_stat_key": {
                "0": true
        },
        "gw_di": {
                "abi": 0,
                "id": "ebb3a0bdf3730aa501snnl",
                "swv": "1.0.3",
                "bv": "40.00",
                "pv": "2.3",
                "lpv": "3.5",
                "pk": "7grflnujaaibcws3",
                "firmk": null,
                "cadv": "1.0.5",
                "cdv": "1.0.0",
                "dev_swv": "1.0.3",
                "s_id": "dw506g",
                "dtp": 0,
                "sync": 0,
                "attr_num": 1,
                "mst_tp_0": 9,
                "mst_ver_0": "1.0.3",
                "mst_md5_0": null,
                "mst_tp_1": 0,
                "mst_ver_1": null,
                "mst_md5_1": null,
                "mst_tp_2": 0,
                "mst_ver_2": null,
                "mst_md5_2": null,
                "mst_tp_3": 0,
                "mst_ver_3": null,
                "mst_md5_3": null,
                "mst_tp_4": 0,
                "mst_ver_4": null,
                "mst_md5_4": null,
                "mst_tp_5": 0,
                "mst_ver_5": null,
                "mst_md5_5": null,
                "mst_tp_6": 0,
                "mst_ver_6": null,
                "mst_md5_6": null,
                "mst_tp_7": 0,
                "mst_ver_7": null,
                "mst_md5_7": null,
                "mst_tp_8": 0,
                "mst_ver_8": null,
                "mst_md5_8": null,
                "mst_tp_9": 0,
                "mst_ver_9": null,
                "mst_md5_9": null
        },
        "wf_start_md": 4,
        "gw_wsm": {
                "nc_tp": 9,
                "ssid": "[redacted]",
                "passwd": "[redacted]",
                "md": 3,
                "wfb64": 1,
                "stat": 2,
                "token": "H8NIrxgp",
                "region": "AZ",
                "reg_key": "_AuR",
                "dns_prio": 0,
                "is_psk30_cfg": true
        },
        "rcs.active": "HEX:0101000002010001030800555 32d50524f440004120068332d617a2e696f742d646e732e636f6d",
```

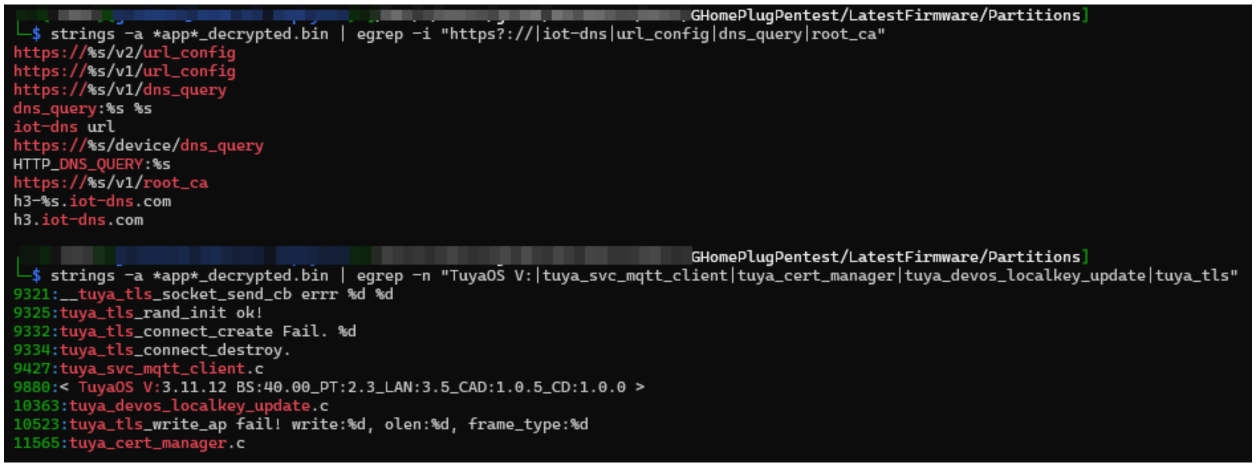
Now this WiFi network is my lab network, but to be safe I am covering the SSID and password which are both encrypted with measly base64-encoding.

Some other interesting details that were returned from some strings analysis:



```
┌─[redacted]GHomePlugPentest/LatestFirmware/Partitions]
└─$ strings -a *app*_decrypted.bin | egrep -i "TUYA_TLS|pskKey|psk key was|TLS-PSK|mqtt|mqtts|local key|auth_key|uuid|psk_id|psk_identity"
mqtt is connected!
mqtt.con
__tuya_tls_socket_send_cb errr %d %d
tuya_tls_rand_init ok!
tuya_tls_connect_create Fail. %d
tuya_tls_connect_destroy.
TUYA_TLS Begin Connect %s:%d
TUYA_TLS PSK Mode
TUYA_TLS Success Connect %s:%d Suit:%s
TUYA_TLS faild Connect %s:%d
TUYA_TLS Disconnect ENTER
TUYA_TLS Disconnect Success
notify: mqtt is blocked
undefined app mqtt protocol:%u, name:%s
mqtt connect deny times:%d
mqtt disconnected
mqtt.disc
mqtt start fail, stat:%d
MQTT Protocol URL:%s Port:%d
mqtt url null
create_mqtt_hand err:%d
mqtt send topic:%s
mqtt-media send topic:%s
Send MQTT Msg.P:%d N:%d Q:%d Data:%s
Send MQTT Msg.P:%d N:%d Q:%d
mqtt-media publish msg. topic:%s
mqtt-media v2 publish msg. topic:%s
MQTT SEND ERR:%d
old key is success. notify mqtt
Rev MQTT:%s
[%s] mqtt state change %d -> %d
mqtt_ping -->>
mqtt_ping err:%d
mqtt_ping <<--
tuya_svc_mqtt_client.c
mqtt close
mqtt fail_cnt:%u, sleeptime:%d ms
mqtt_thread paused
connect to mqtt broker %s port %d
mqtt Get IP Fails %d
mqtt client ip:%s, link-tp:%d
mqtts
setup mqtt transporter success
mqtt connect ret:%d.
send mqtt connect success
host[%s] mqtt connect resp err:%d
mqtt topics cnt %d
send mqtt subscribe success
close backup mqtt transporter
mqtt recv err %d
mqtt_thread exit
mqtts://%s
mqtt_thread
start ble v2 encrpt by mqtt
UUID invalid.
upgrade.mqtt.notify
mqtt report download percent:%d
uuid authkey invalid
auth_key
wx_uuid
pskKey
gw_pskkey_get err:%d
pskKey too long:%d
{"ip":"%s","gwId":"%s","uuid":"%s"
psk_identity
tuya_tls_write_ap fail! write:%d, olen:%d, frame_type:%d
emqtt
```

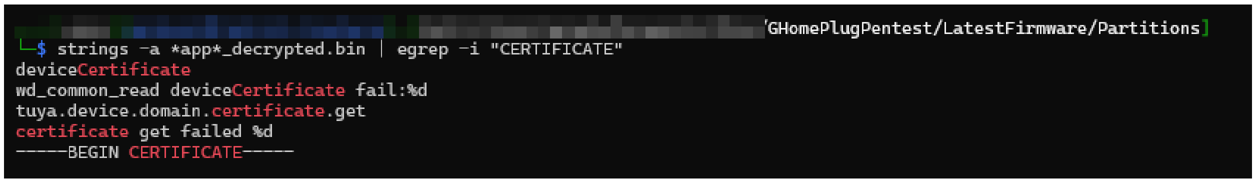
I have taken some time to try and investigate some of the details regarding the auth_key details, and the tls strings for encryption details. Truthfully, I didn't get far and decided to take another route that I will discuss later.



```
[redacted]GHomePlugPentest/LatestFirmware/Partitions]
└─$ strings -a *app*_decrypted.bin | egrep -i "https?://|iot-dns|url_config|dns_query|root_ca"
https://%s/v2/url_config
https://%s/v1/url_config
https://%s/v1/dns_query
dns_query:%s %s
iot-dns url
https://%s/device/dns_query
HTTP_DNS_QUERY:%s
https://%s/v1/root_ca
h3-%s.iot-dns.com
h3.iot-dns.com

[redacted]GHomePlugPentest/LatestFirmware/Partitions]
└─$ strings -a *app*_decrypted.bin | egrep -n "TuyaOS V:|tuya_svc_mqtt_client|tuya_cert_manager|tuya_devos_localkey_update|tuya_tls"
9321:__tuya_tls_socket_send_cb errr %d %d
9325:tuya_tls_rand_init ok!
9332:tuya_tls_connect_create Fail. %d
9334:tuya_tls_connect_destroy.
9427:tuya_svc_mqtt_client.c
9880:< TuyaOS V:3.11.12 BS:40.00_PT:2.3_LAN:3.5_CAD:1.0.5_CD:1.0.0 >
10363:tuya_devos_localkey_update.c
10523:tuya_tls_write_ap fail! write:%d, olen:%d, frame_type:%d
11565:tuya_cert_manager.c
```

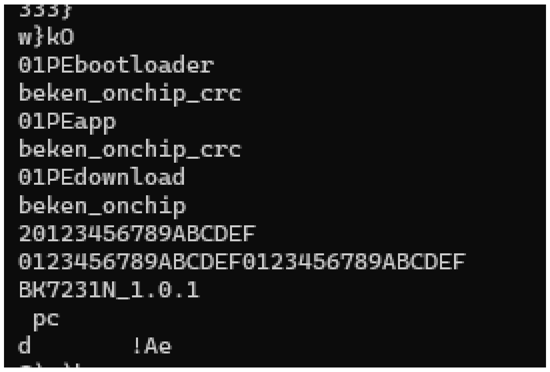
We get some domains of interest here, as well as a program to look for when approaching the means of encryption on the device, ‘tuya_cert_manager.c’. This will be something I will have to look at in the future.

There also seems to be a certificate, but I was unable to find this certificate using pure heuristics of the strings within. There is also no ‘END’ label for the cert.

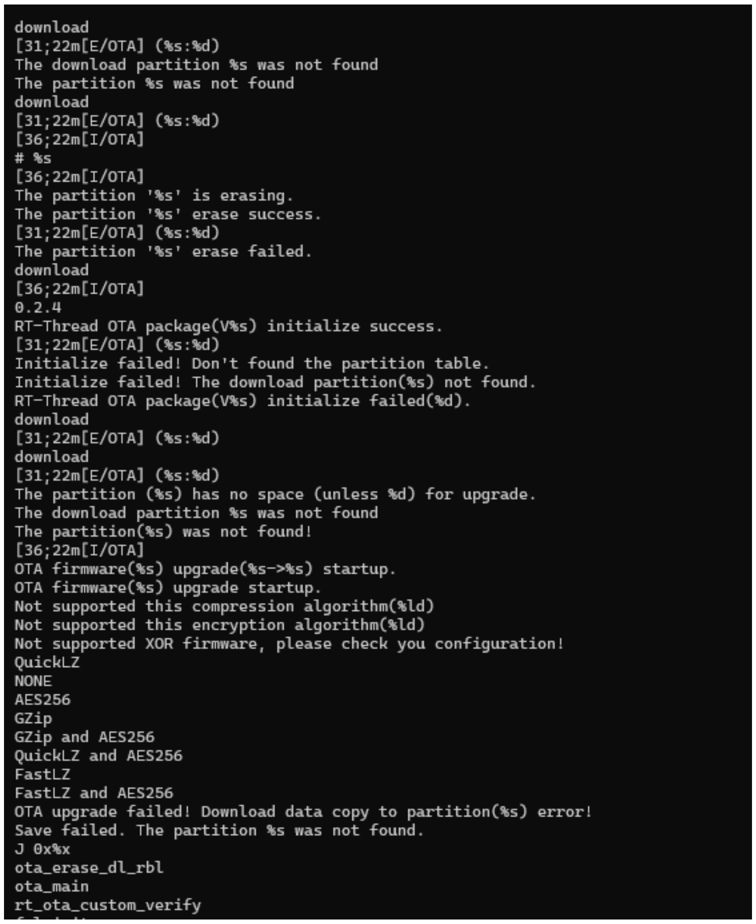


```
[redacted]'GHomePlugPentest/LatestFirmware/Partitions]
└─$ strings -a *app*_decrypted.bin | egrep -i "CERTIFICATE"
deviceCertificate
wd_common_read deviceCertificate fail:%d
tuya.device.domain.certificate.get
certificate get failed %d
-----BEGIN CERTIFICATE-----
```

While looking at the bootloader strings we find two important artifacts: the version and some strings regarding OTA logging (Over The Air) pertaining to remote (Over The Air) updates.



```
333}
w}kO
01PEbootloader
beken_onchip_crc
01PEapp
beken_onchip_crc
01PEdownload
beken_onchip
20123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
BK7231N_1.0.1
 pc
d        !Ae
```


```
download
[31;22m[E/OTA] (%s:%d)
The download partition %s was not found
The partition %s was not found
download
[31;22m[E/OTA] (%s:%d)
[36;22m[I/OTA]
# %s
[36;22m[I/OTA]
The partition '%s' is erasing.
The partition '%s' erase success.
[31;22m[E/OTA] (%s:%d)
The partition '%s' erase failed.
download
[36;22m[I/OTA]
0.2.4
RT-Thread OTA package(V%s) initialize success.
[31;22m[E/OTA] (%s:%d)
Initialize failed! Don't found the partition table.
Initialize failed! The download partition(%s) not found.
RT-Thread OTA package(V%s) initialize failed(%d).
download
[31;22m[E/OTA] (%s:%d)
download
[31;22m[E/OTA] (%s:%d)
The partition (%s) has no space (unless %d) for upgrade.
The download partition %s was not found
The partition(%s) was not found!
[36;22m[I/OTA]
OTA firmware(%s) upgrade(%s->%s) startup.
OTA firmware(%s) upgrade startup.
Not supported this compression algorithm(%ld)
Not supported this encryption algorithm(%ld)
Not supported XOR firmware, please check you configuration!
QuickLZ
NONE
AES256
GZip
GZip and AES256
QuickLZ and AES256
FastLZ
FastLZ and AES256
OTA upgrade failed! Download data copy to partition(%s) error!
Save failed. The partition %s was not found.
J 0x%x
ota_erase_dl_rbl
ota_main
rt_ota_custom_verify
```

To note, we can assume AES, GZIP, and QuickLZ are likely used somewhere within the device.

Different Wireshark Captures

Now to this point I have been mostly linear in my explanation, but there's really been multiple runs of extracting firmware, doing some analysis, then setting up more on the device. These different steps went:

- Out of box
- Connecting to local WiFi
- Connecting to WiFi and using app to run commands (therefore initiating communication with the cloud server.)

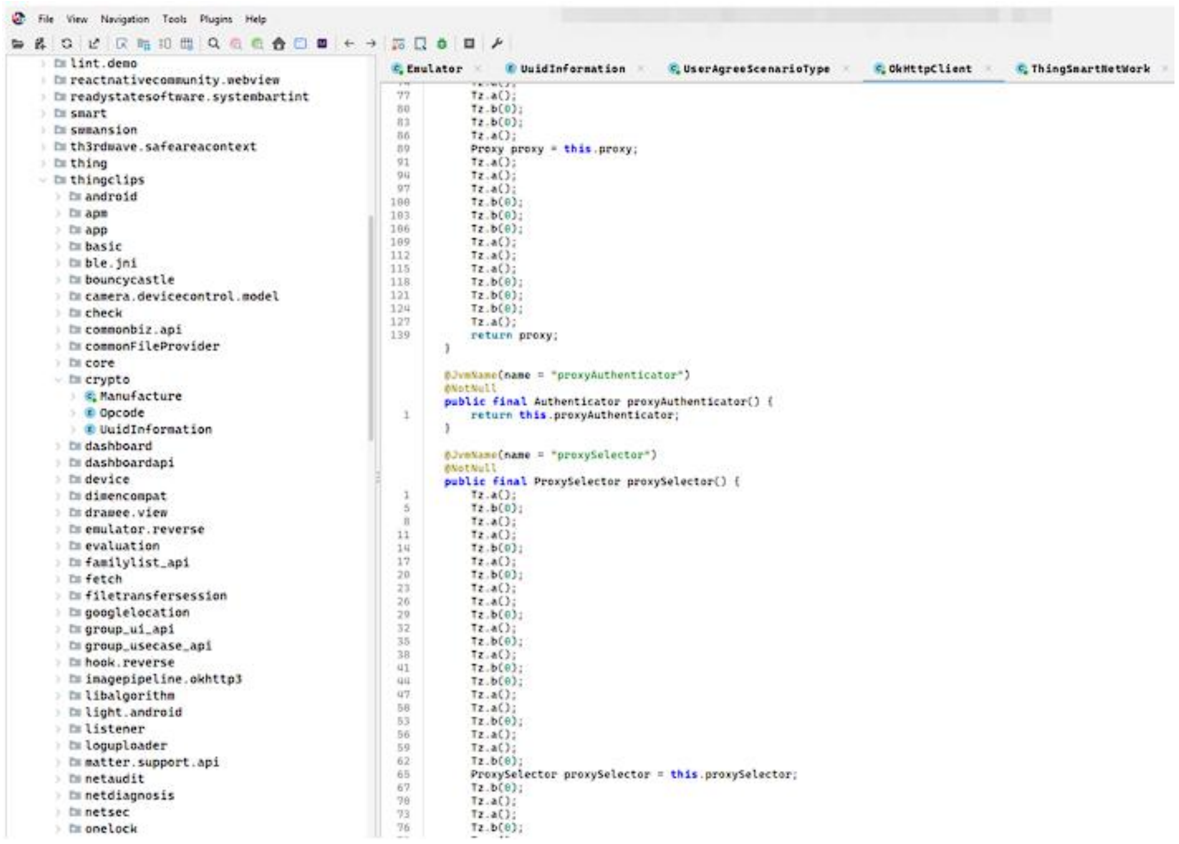
All of these Wireshark transmissions gave me some useful data in regard to domains, but we kind of knew that before (yes I respect verification, but we'll get to that).

Deciding to look at the application

At this point I devoted around half a day to trying to modify the firmware to have a new root of trust for my own proxy or at least extract its certs. Then, I remembered: this thing has a whole app that I control. I realized that I could just add a root of trust to my proxy onto my phone and then run the app to capture traffic. I will also say that I downloaded the .APK from here:

Download Gosund APKs for Android - APKMirror

I opened up the .apk in a tool that a recent co-worker of mine showed me this past summer, JADX (If that coworker says it's okay to add his LinkedIn then I will do that later).



```
lint.demo
reactnativecommunity.webview
readystatesoftware.systembartint
smart
swmansion
th3rdwave.safeareacontext
thing
thingclips
  android
  apm
  app
  basic
  ble.jni
  bouncycastle
  camera.devicecontrol.model
  check
  commonbiz.api
  commonFileProvider
  core
  crypto
    Manufacture
    Opcode
    UuidInformation
  dashboard
  dashboardapi
  device
  dimencompat
  dramee.view
  emulator.reverse
  evaluation
  familylist_api
  fetch
  filetransfersession
  googlelocation
  group_ui_api
  group_usecase_api
  hook.reverse
  imagepipeline.okhttp3
  libalgorithm
  light.android
  listener
  loguploader
  matter.support.api
  netaudit
  netdiagnosis
  netsec
  onelock
```

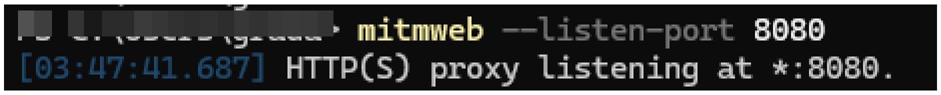
```java
        Tz.a();
        Tz.b(0);
        Tz.b(0);
        Tz.a();
        Proxy proxy = this.proxy;
        Tz.a();
        Tz.a();
        Tz.a();
        Tz.b(0);
        Tz.b(0);
        Tz.b(0);
        Tz.a();
        Tz.a();
        Tz.a();
        Tz.b(0);
        Tz.b(0);
        Tz.b(0);
        Tz.a();
        return proxy;
    }

    @JvmName(name = "proxyAuthenticator")
    @NotNull
    public final Authenticator proxyAuthenticator() {
        return this.proxyAuthenticator;
    }

    @JvmName(name = "proxySelector")
    @NotNull
    public final ProxySelector proxySelector() {
        Tz.a();
        Tz.b(0);
        Tz.a();
        Tz.a();
        Tz.b(0);
        Tz.a();
        Tz.b(0);
        Tz.a();
        Tz.a();
        Tz.b(0);
        Tz.a();
        Tz.b(0);
        Tz.a();
        Tz.b(0);
        Tz.b(0);
        Tz.a();
        Tz.a();
        Tz.b(0);
        Tz.a();
        Tz.a();
        Tz.b(0);
        ProxySelector proxySelector = this.proxySelector;
        Tz.b(0);
        Tz.a();
        Tz.a();
        Tz.b(0);
```

This may not be funny to most people (or surprising), but this thing was pretty heavily obfuscated: even to the point where there would be hundreds of function calls and those functions don't do anything. Function names, of course, were also not anywhere near what they did. I did take part in some java decomp analysis which will be a path I take more in the future, but the reasoning I pivoted is because I realized it would take me a long time to handle the encryption being done. Not an impossible task, and it is the next step I will be taking.

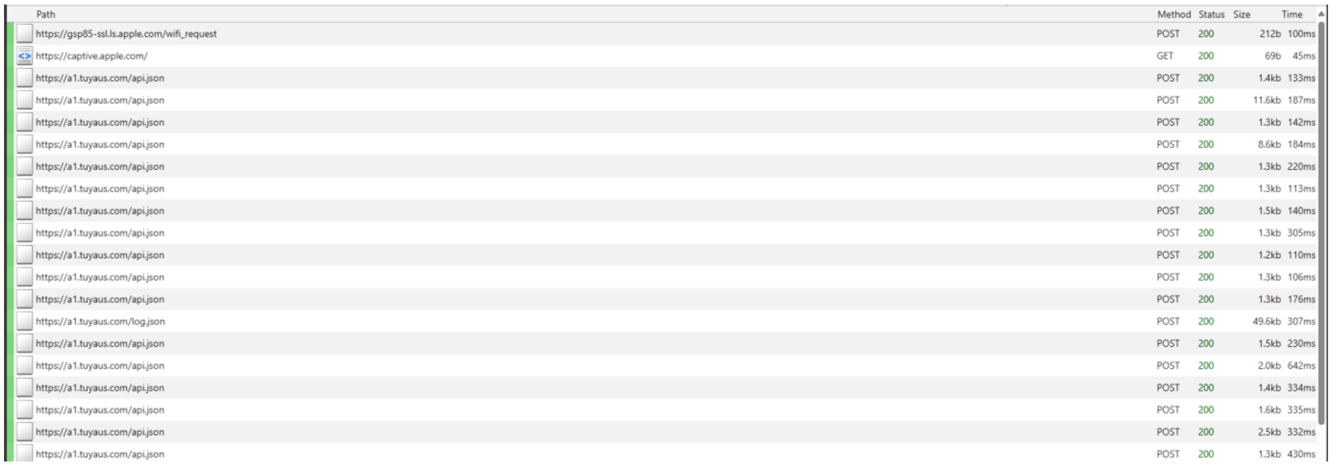
Setting up the proxy

Many people have their own preference in regards to how to analyze proxy traffic, but personally I enjoy the mitm utility, 'mitmweb'. So I configured my phone to trust the mitm proxy cert and then started a mitmweb instance. Directions to set up the cert can be found here:

[Setting up mitmproxy with iOS 17.1 – Trickster Dev]()



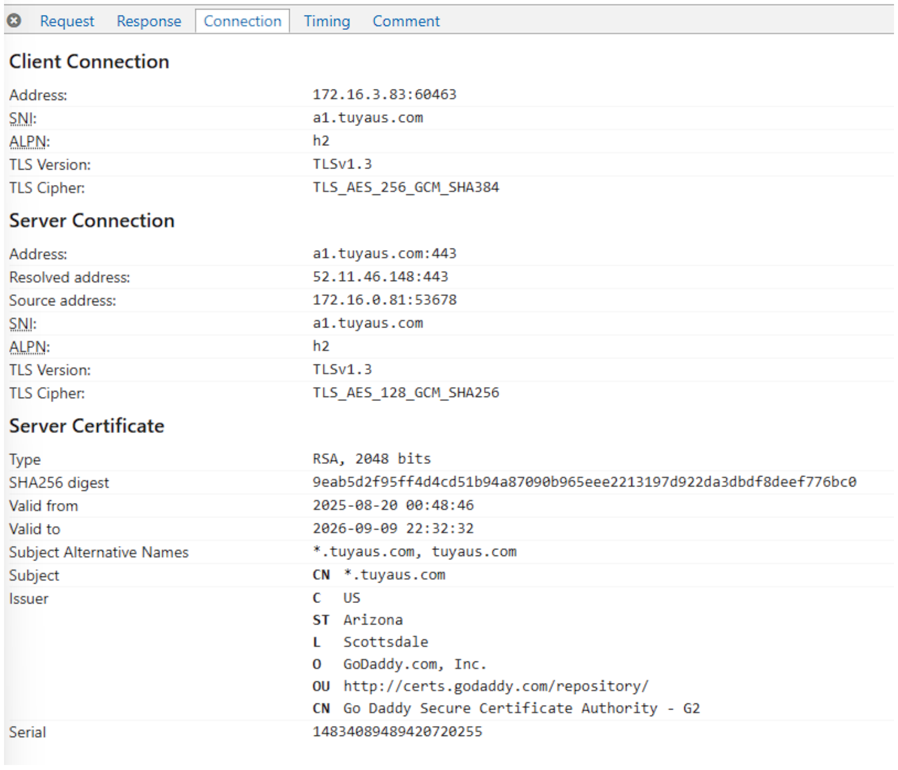
```
[redacted] mitmweb --listen-port 8080
[03:47:41.687] HTTP(S) proxy listening at *:8080.
```



Path	Method	Status	Size	Time
https://gsp85-ssl.ls.apple.com/wifi_request	POST	200	212b	100ms
https://captive.apple.com/	GET	200	69b	45ms
https://a1.tuyaus.com/api.json	POST	200	1.4kb	133ms
https://a1.tuyaus.com/api.json	POST	200	11.6kb	187ms
https://a1.tuyaus.com/api.json	POST	200	1.3kb	142ms
https://a1.tuyaus.com/api.json	POST	200	8.6kb	184ms
https://a1.tuyaus.com/api.json	POST	200	1.3kb	220ms
https://a1.tuyaus.com/api.json	POST	200	1.3kb	113ms
https://a1.tuyaus.com/api.json	POST	200	1.5kb	140ms
https://a1.tuyaus.com/api.json	POST	200	1.3kb	305ms
https://a1.tuyaus.com/api.json	POST	200	1.2kb	110ms
https://a1.tuyaus.com/api.json	POST	200	1.3kb	106ms
https://a1.tuyaus.com/api.json	POST	200	1.3kb	176ms
https://a1.tuyaus.com/log.json	POST	200	49.6kb	307ms
https://a1.tuyaus.com/api.json	POST	200	1.5kb	230ms
https://a1.tuyaus.com/api.json	POST	200	2.0kb	642ms
https://a1.tuyaus.com/api.json	POST	200	1.4kb	334ms
https://a1.tuyaus.com/api.json	POST	200	1.6kb	335ms
https://a1.tuyaus.com/api.json	POST	200	2.5kb	332ms
https://a1.tuyaus.com/api.json	POST	200	1.3kb	430ms

Once I start getting plaintext requests, I can analyze how the app is communicating with the cloud server in order to control the device.

We confirm the domain
‘a1.tuyaus.com’ using the IP address ’52.11.46.148’ is being used by the clo
ud server.



Client Connection

Address:	172.16.3.83:60463
SNI:	a1.tuyaus.com
ALPN:	h2
TLS Version:	TLSv1.3
TLS Cipher:	TLS_AES_256_GCM_SHA384

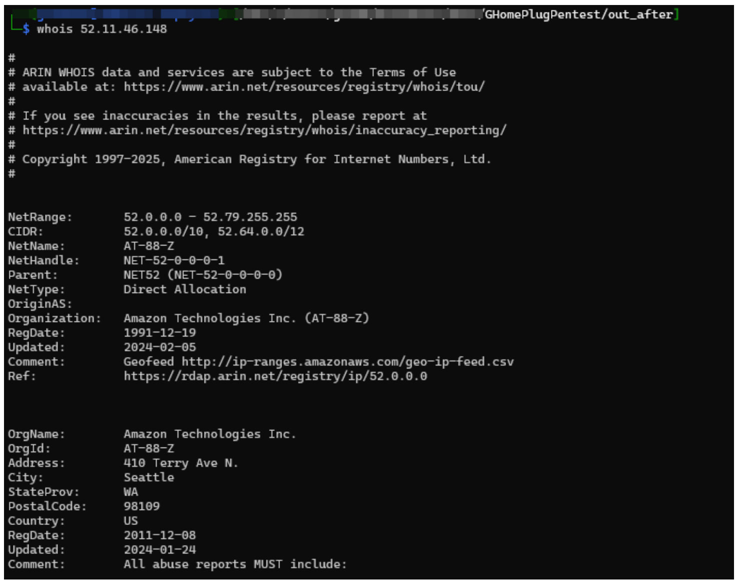
Server Connection

Address:	a1.tuyaus.com:443
Resolved address:	52.11.46.148:443
Source address:	172.16.0.81:53678
SNI:	a1.tuyaus.com
ALPN:	h2
TLS Version:	TLSv1.3
TLS Cipher:	TLS_AES_128_GCM_SHA256

Server Certificate

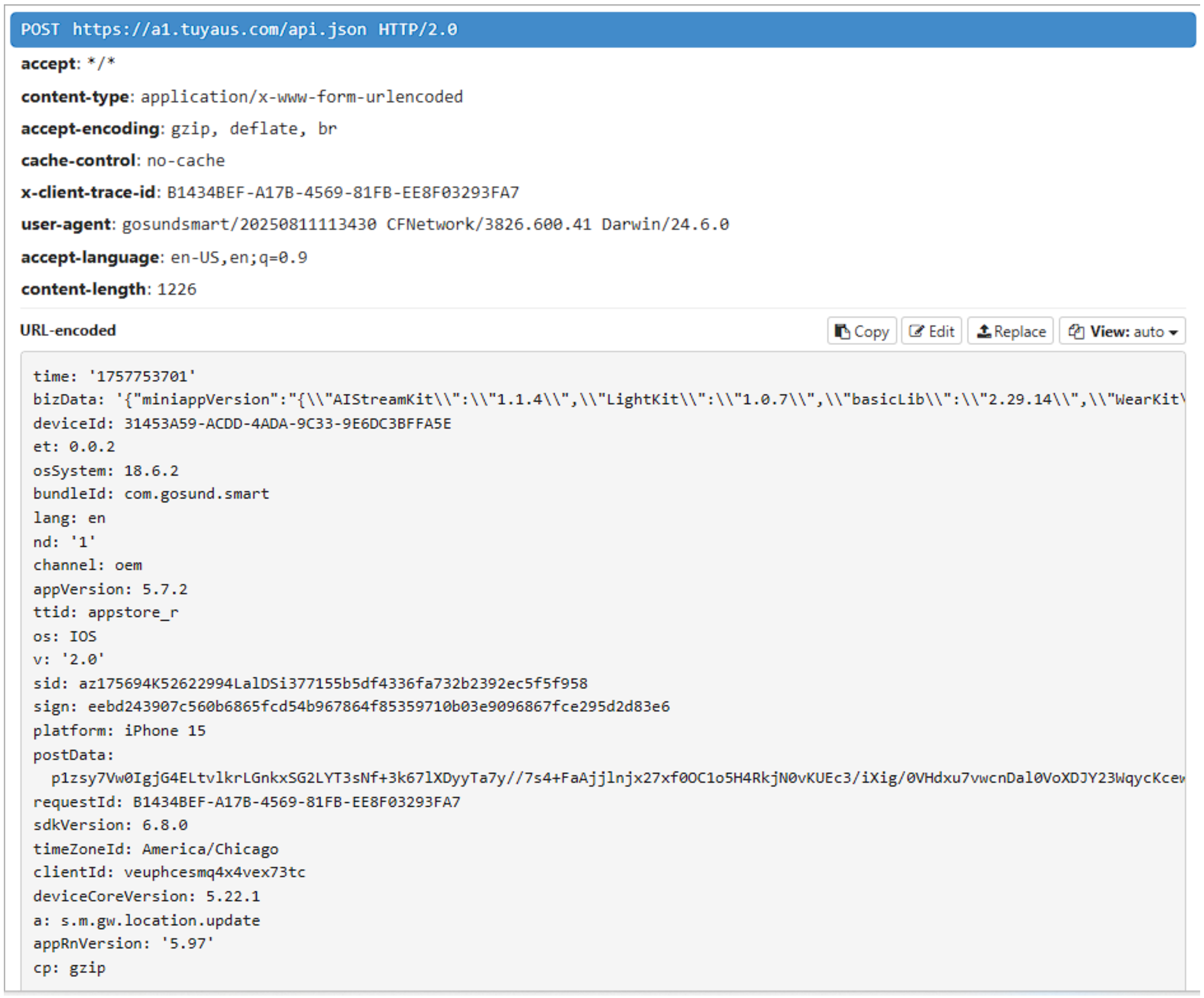
Type	RSA, 2048 bits
SHA256 digest	9eab5d2f95ff4d4cd51b94a87090b965eee2213197d922da3dbdf8deef776bc0
Valid from	2025-08-20 00:48:46
Valid to	2026-09-09 22:32:32
Subject Alternative Names	*.tuyaus.com, tuyaus.com
Subject	**CN** *.tuyaus.com
Issuer	**C** US **ST** Arizona **L** Scottsdale **O** GoDaddy.com, Inc. **OU** http://certs.godaddy.com/repository/ **CN** Go Daddy Secure Certificate Authority - G2
Serial	14834089489420720255

This IP is owned by amazon, so it is likely an AWS cloud instance that their server is running on. That said, it should be noted that there’s no guarantee that the traffic isn’t then forwarded to a Chinese server. This redirection of traffic is often used to prevent block lists geared towards preventing traffic to China from working. (Think an IT person has all Chinese IP addresses blocked on his firewall, this device would show that it’s only sending the data to an American IP.)



```
┌──[redacted]GHomePlugPentest/out_after]
└─$ whois 52.11.46.148

#
# ARIN WHOIS data and services are subject to the Terms of Use
# available at: https://www.arin.net/resources/registry/whois/tou/
#
# If you see inaccuracies in the results, please report at
# https://www.arin.net/resources/registry/whois/inaccuracy_reporting/
#
# Copyright 1997-2025, American Registry for Internet Numbers, Ltd.
#


NetRange:       52.0.0.0 - 52.79.255.255
CIDR:           52.0.0.0/10, 52.64.0.0/12
NetName:        AT-88-Z
NetHandle:      NET-52-0-0-0-1
Parent:         NET52 (NET-52-0-0-0-0)
NetType:        Direct Allocation
OriginAS:
Organization:   Amazon Technologies Inc. (AT-88-Z)
RegDate:        1991-12-19
Updated:        2024-02-05
Comment:        Geofeed http://ip-ranges.amazonaws.com/geo-ip-feed.csv
Ref:            https://rdap.arin.net/registry/ip/52.0.0.0


OrgName:        Amazon Technologies Inc.
OrgId:          AT-88-Z
Address:        410 Terry Ave N.
City:           Seattle
StateProv:      WA
PostalCode:     98109
Country:        US
RegDate:        2011-12-08
Updated:        2024-01-24
Comment:        All abuse reports MUST include:
```

In regards to the traffic itself, a typical request looks something like this:



```
POST https://a1.tuyaus.com/api.json HTTP/2.0
accept: */*
content-type: application/x-www-form-urlencoded
accept-encoding: gzip, deflate, br
cache-control: no-cache
x-client-trace-id: B1434BEF-A17B-4569-81FB-EE8F03293FA7
user-agent: gosundsmart/20250811113430 CFNetwork/3826.600.41 Darwin/24.6.0
accept-language: en-US,en;q=0.9
content-length: 1226
```

URL-encoded

```
time: '1757753701'
bizData: '{"miniappVersion":"{\\"AIStreamKit\\":\\"1.1.4\\",\\"LightKit\\":\\"1.0.7\\",\\"basicLib\\":\\"2.29.14\\",\\"WearKit\
deviceId: 31453A59-ACDD-4ADA-9C33-9E6DC3BFFA5E
et: 0.0.2
osSystem: 18.6.2
bundleId: com.gosund.smart
lang: en
nd: '1'
channel: oem
appVersion: 5.7.2
ttid: appstore_r
os: IOS
v: '2.0'
sid: az175694K52622994LalDSi377155b5df4336fa732b2392ec5f5f958
sign: eebd243907c560b6865fcd54b967864f85359710b03e9096867fce295d2d83e6
platform: iPhone 15
postData:
  p1zsy7Vw0IgjG4ELtvlkrLGnkxSG2LYT3sNf+3k67lXDyyTa7y//7s4+FaAjjlnjx27xf0OC1o5H4RkjN0vKUEc3/iXig/0VHdxu7vwcnDal0VoXDJY23WqycKcew
requestId: B1434BEF-A17B-4569-81FB-EE8F03293FA7
sdkVersion: 6.8.0
timeZoneId: America/Chicago
clientId: veuphcesmq4x4vex73tc
deviceCoreVersion: 5.22.1
a: s.m.gw.location.update
appRnVersion: '5.97'
cp: gzip
```

To break down my current thoughts, this is what I believe each field means thus far:

- **time: '1757753701'**: Unix timestamp
- **bizData**: JSON string containing app component versions & flags:
  - **miniappVersion** → lists the versions of internal SDK kits (AIStreamKit, LightKit, MapKit, etc.). These are modular libraries that Tuya apps load dynamically for IoT, AI, video, etc.
  - **nd** → maybe this means *"new device"* or *"network domain"*.
  - **customDomainSupport** → "1" means the app supports overriding default Tuya cloud domains (e.g., for regional clouds or private Tuya deployments).
- **deviceId: 31453A59-ACDD-4ADA-9C33-9E6DC3BFFA5E**

  A UUID generated on the iOS device to uniquely identify it to Tuya's backend.
- **et: 0.0.2**: Event type or envelope version? Idrk.
- **osSystem: 18.6.2** OS version (iOS 18.6.2 in this case).
- **bundleId: com.gosund.smart**

  iOS app bundle ID (identifies the app — Gosund Smart, which is Tuya-based).
- **lang: en** → Language set in the app (English).
- **nd: '1'** → Same thing repeated at the envelope level
- **channel: oem**: Distribution channel. "oem" = this is an OEM-branded app (Gosund, GHome, etc.), not Tuya's stock Smart Life. I saw some remnants of this when decompiling the app.
- **appVersion: 5.7.2**: The app version installed on the phone.

- **ttid: appstore_r**: Likely “Tuya Tracking ID” → “appstore release”. Used internally for tracking distribution source (App Store, TestFlight, etc.).
- **os: IOS**: Operating system name.
- **v: '2.0'**: API protocol version of this request format.
- **sid: az175694K52622994LalDSi377155b5df4336fa732b2392ec5f5f958**: Session ID (auth token) issued after login. Identifies a user’s authenticated session with Tuya cloud.
- **sign: eebd243907c560b6865fcd54b967864f85359710b03e9096867fce295d2d83e6**: HMAC-style signature of the request (usually SHA-256 over fields like time, bizData, sid, and clientId, combined with the device’s **auth_key**). Prevents tampering.
- **platform: iPhone 15**: Device model.
- **postData:** Encrypted business payload (often gzip + AES/PSK encrypted). The big base64 string is the actual request body (e.g., location update, control command). Only the Tuya SDK and server can decode it with the session’s negotiated keys. This is why I am going to target the SDK in order to decrypt this for packets in the future.

- **requestId: B1434BEF-A17B-4569-81FB-EE8F03293FA7**

UUID for this request. Lets Tuya correlate client/server logs.

- **sdkVersion: 6.8.0**: Version of the Tuya mobile SDK embedded in the app.
- **timeZoneId: America/Chicago**: Device timezone setting.
- **clientId: veuphcesmq4x4vex73tc**

App’s registered client identifier (different from deviceId). Part of Tuya’s OAuth-style API authentication.

- **deviceCoreVersion: 5.22.1**:

 Version of Tuya’s “core device service” inside the SDK — the layer that handles device binding, MQTT, encryption, etc.

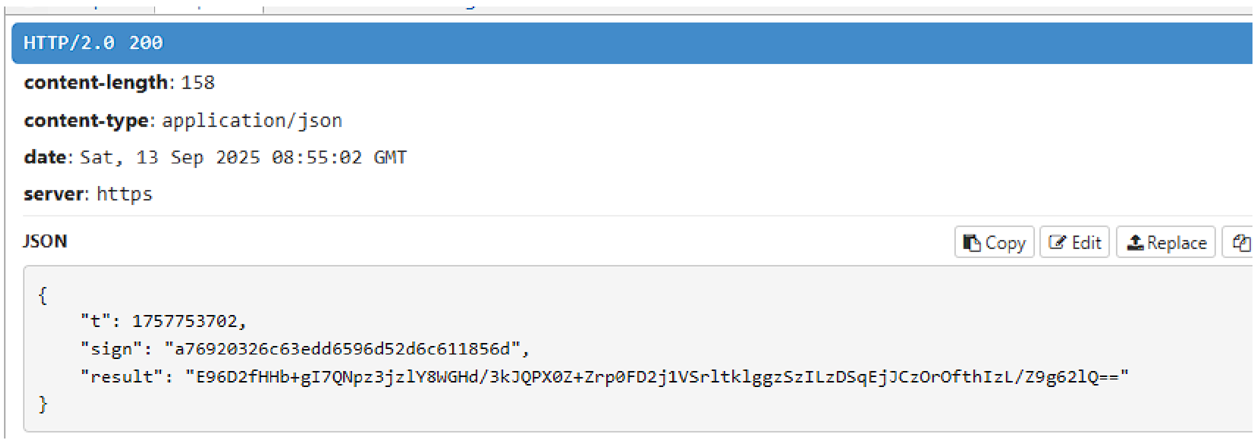
- **a: s.m.gw.location.update**:

 API method being called. In this case: “service.mobile.gateway.location.update” → means the app is updating the phone’s (or device’s) location info with Tuya Cloud.

- **appRnVersion: '5.97'**:

Version of the React Native framework bundle used in the app. Tuya apps embed a hybrid RN shell.

- **cp: gzip**: Compression flag. Tells the server that postData was compressed with gzip before encryption. We will need to use this fact in the future in order to get the details within postData.



```
HTTP/2.0 200
content-length: 158
content-type: application/json
date: Sat, 13 Sep 2025 08:55:02 GMT
server: https
```

```
{
    "t": 1757753702,
    "sign": "a76920326c63edd6596d52d6c611856d",
    "result": "E96D2fHHb+gI7QNpz3jzlY8WGHd/3kJQPX0Z+Zrp0FD2j1VSrltklggzSzILzDSqEjJCzOrOfthIzL/Z9g621Q=="
}
```

It is also worth mentioning that this is a typical response to the post request.

t: 1757753702

- Unix timestamp from the server
- Used to prevent replay attacks.

sign: "a76920326c63edd6596d52d6c611856d"

- Signature generated by the server over its response payload.
- Maybe an **MD5 or SHA-256 hash** of (result + secret/session key + t) .
- The client will recompute the sign using its known auth_key or session key, and compare. If it mismatches, the packet is discarded.

result: "E96D2fHHb+glTQNp...Z9g62lQ="

- Base64-encoded blob.
- Inside: compressed (gzip) + encrypted (AES/PSK negotiated during TLS-PSK handshake).
- Contains the actual **business data** (e.g., device control result, location acknowledgment, or configuration info).
- Only the SDK should be able to decrypt with the active session keys (derived from pskKey/auth_key).

Closing Remarks and Future Work

This analysis of the GHome/Tuya smart plug shows how much useful information can be extracted from a low-cost consumer IoT device using only UART access, firmware partitioning, and controlled network observation. From identifying the SoC and bypassing the ROM boot protections, to separating partitions and capturing live TLS-PSK traffic, I had a great time in learning more about the process of how I want to approach problems like this one. I think in the future I will likely make the jump to decrypting TLS traffic from the application side first, so that I have some more details on what I want to look into in regards to the binary's.

The next phase of this work will focus on deeper application analysis. By unpacking and reversing the Gosund APK, I plan to identify how the Tuya SDK derives request signatures and encrypts the postData payloads. If the decryption routines can be replicated outside the app, I will be able to inspect the true contents of these cloud API calls. This will also allow testing whether endpoint authorization checks are robust — specifically, verifying whether one authenticated device can issue commands to control another device under the same or different accounts. Since I purchased a second plug for controlled testing, this presents a safe way to validate cross-device command isolation and ensure Tuya's APIs enforce proper device ownership. I will also investigate the Bluetooth interface of the device, as I haven't investigated that attack vector yet. Lastly, I want to connect it to another home assistant device (say Alexa), and see what data is being sent between the two.

In short, this project has demonstrated the value of combining hardware access with software and network analysis. Future work will be dedicated to bridging that gap: decompiling the APK, reconstructing the crypto routines, and validating whether Tuya's backend APIs truly enforce strong isolation and access control. I know this writeup is long, and I hope at least one person can get something positive out of it, because I did.

As always:

May the force be with you, God speed,

And God bless.