Exploiting TP-Link HS110 Smart Plug

ECE419/CS460 Final Project
Spring 2019

Rishov Dutta (rsdutta2), Daniel Zhang(dzhang54)

Introduction

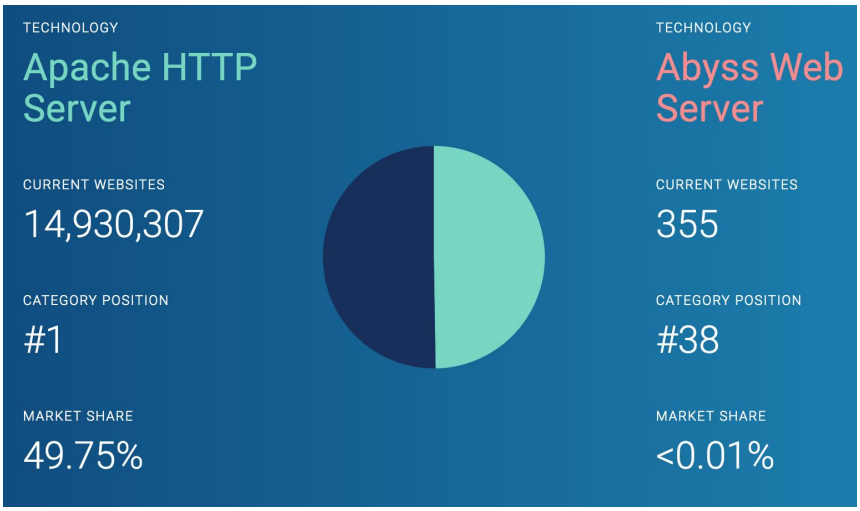
The TPLink-HS110 is a wifi enabled smart plug which allows users to to turn an electrical outlet on or off from anywhere on the internet. This device has its own wireless chip and features an open port which allows it to communicate to users via an app. This makes any form of attack more complicated since direct commands seem to have to go through the app on first look. We investigated how exactly the how the smart plug communicates with the app in the local network as well as remotely. The results of our analysis is shown below.

Setup and initial portscan

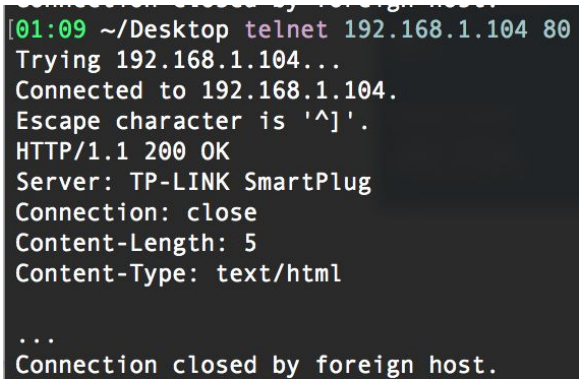
After setting up the smart plug, we used nmap to to do a portscan. We show the results here.

```
[00:45 ~/Desktop nmap 192.168.1.104

Starting Nmap 7.60 ( https://nmap.org ) at 2019-04-27 00:45 CDT
Nmap scan report for 192.168.1.104
Host is up (0.013s latency).
Not shown: 998 closed ports
PORT     STATE SERVICE
80/tcp   open  http
9999/tcp open  abyss

Nmap done: 1 IP address (1 host up) scanned in 6.95 seconds
```

As seen above, the portscan shows that both ports 80 and 9999 are open accepting TCP protocols running http and abyss services. We were familiar with http from this class as well as security 1, but did not know what abyss means. After further searching through stackoverflow, we learned that abyss is a web server, similar to an Apache http server. However, abyss has significantly fewer users, as shown in the picture below, so there may be vulnerabilities we can exploit here.

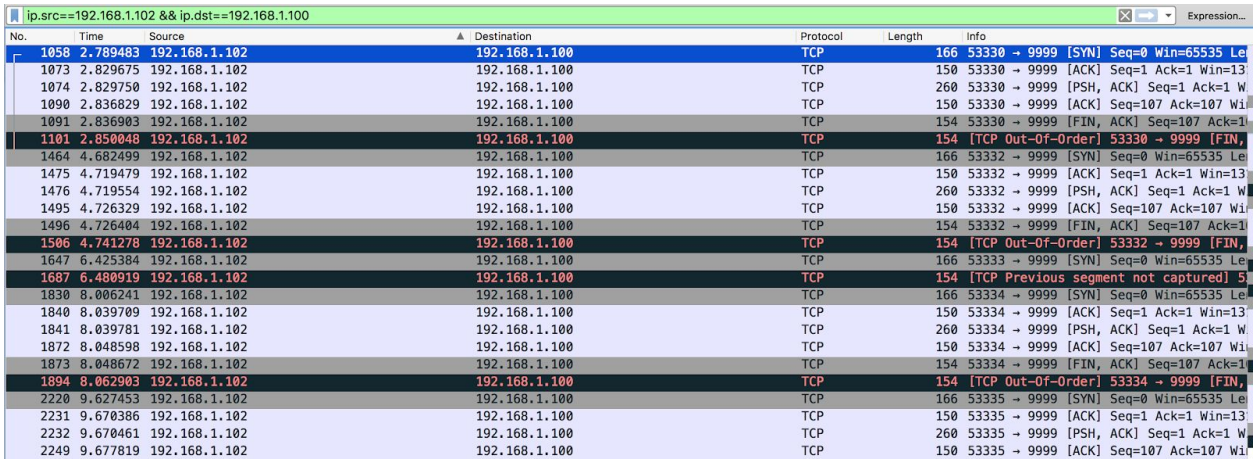


We first looked to see what would happen if we connected to port 80. We did this by performing a telnet command, but we got a garbage response. We attempted this again just by searching in the browser, but got the same result.



Wireshark analysis

We then shifted our attention to port 9999 running the abyss webserver. Telnet just gave us a connection closed and the browser just returned no response from server. It seemed that we could not just connect and write commands to the smart plug. We needed to find out how it received commands. To do this, we used the app on the local network and used wireshark to inspect the packets that were sent. This proved to be somewhat of a challenge because in order to decrypt the capture, we needed to obtain the four-way handshake from the router because we configured the network to use wpa encryption. Our first solution was to just make the network open, but the kasa app would not allow us to connect to an open network, which is a nice security feature. Retrieving the handshake was difficult at first because we had never done this before, but after searching online, we were able to obtain it and decrypt the capture. Below is the wireshark capture, filtered only for communication between the app (on the phone) and the smart plug.



The phone communicates to the app by sending packets to port 9999 on the smart plug directly. Below are the hex outputs of the payload.

Off(app):

```
00:00:00:66:d0:f2:81:f8:8b:ff:9a:f7:d5:ef:94:b6:c5:a0:d4:8b:f9:9c:f0:91:e8:b7:c4:b0:d1:a5:c0:e2:d8:a3:81:f2:86:
e7:93:f6:d4:ee:de:a3:de:f2:d0:b3:dc:b2:c6:a3:db:af:8d:b7:cc:ee:9d:f2:87:f5:96:f3:d1:eb:c9:ab:c8:ad:c9:fd:98:fd:
ce:e3:80:e6:d3:e7:ca:fe:c9:ff:c8:e5:dc:ec:de:bf:92:a6:93:f1:90:f2:c4:f7:c6:f5:c3:a0:97:b5:c8:b5
```

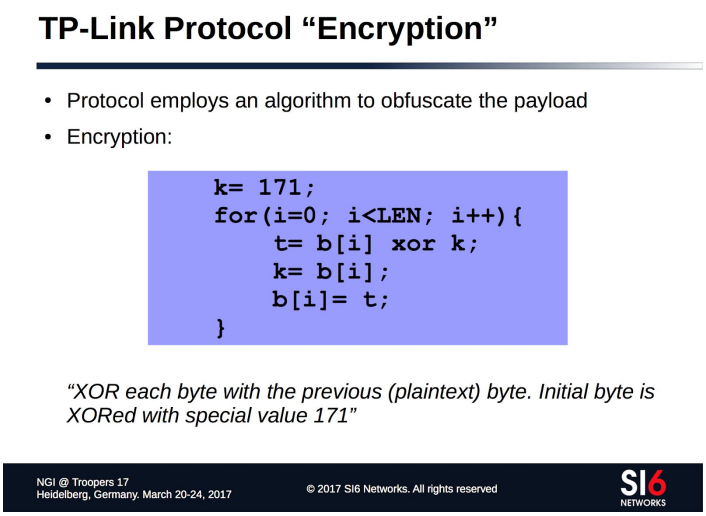
On(app):

```
00:00:00:66:d0:f2:81:f8:8b:ff:9a:f7:d5:ef:94:b6:c5:a0:d4:8b:f9:9c:f0:91:e8:b7:c4:b0:d1:a5:c0:e2:d8:a3:81:f2:86:
e7:93:f6:d4:ee:df:a2:df:f3:d1:b2:dd:b3:c7:a2:da:ae:8c:b6:cd:ef:9c:f3:86:f4:97:f2:d0:ea:c8:aa:c9:ac:c8:fc:99:fc:
cf:e2:81:e7:d2:e6:cb:ff:c8:fe:c9:e4:dd:ed:df:be:93:a7:92:f0:91:f3:c5:f6:c7:f4:c2:a1:96:b4:c9:b4
```

We spammed the on/off button a few times and noticed the payloads would be exactly the same. Upon further inspection, we noticed that all the payload for off and on were almost identical as well. Off ends with b5 and on ends with b4 (first photo above is off, second is on). We saw that the payloads start to differ at the 43rd byte, where off corresponds to DE and on corresponds to DF. With this information, we wrote a script to turn the smart plug on and off by converting the payload to bytes and just sending them to the smart plug from any device connected locally. We were able to turn the smart plug on and off without using the app. There does not seem to be any security regarding who sends the packets, at least for a local connection.

Encryption

We looked further into the hex dumps and googled common encryption schemes that would fit the sequential nature that we explained above. Every hex number after the first difference was off by one, hence ending in b4 and b5 for off and on respectively. This would fit the bill for a vigenere cipher, but after trying an online decoder, we came up empty handed. After even more googling, we came across a powerpoint which mentions that TP-Link uses the same encryption scheme for all of their smart devices.



We implemented this algorithm in python except reversed and were able to decode messages on the local network; this did not work for any cloud communication to the cloud server from the app.

Remote Connection and More Commands

When we accessed the smart plug remotely, the plug would receive a connection from the TP-Link cloud, but the payloads for on and off were different from those on the local connection. We tried to extract credentials from the application request to the cloud but there is increased security here and just resending the packets to the cloud server did not work, nor did looking for ‘user’ and ‘pass’ fields in the request. We assumed that there must be some sort of checksum, device check, or handshake that takes place as well as more secure encryption, but we were unable to look into it further. If there was a remote cloud vulnerability easy enough for us amateur security students to abuse, TP-Link would have major problems. However, we did find an api which allows users to log in and control their devices remotely. This allowed us to extract the hex dumps from the cloud server to the device for new commands and add them to our script for use later, beyond the on and off from the app. These new commands include seeing how much energy the plug used, the nearby wifi networks, a schedule for the plug, a reboot, and a full factory reset. Their hex dumps are listed below.

```
Off(server)
00:00:00:2a:d0:f2:81:f8:8b:ff:9a:f7:d5:ef:94:b6:c5:a0:d4:8b:f9:9c:f0:91:e8:b7:c4:b0:d1:a5:c0:e2:d8:a3:81:f2:86:
e7:93:f6:d4:ee:de:a3:de:a3

On(Server)
00:00:00:2a:d0:f2:81:f8:8b:ff:9a:f7:d5:ef:94:b6:c5:a0:d4:8b:f9:9c:f0:91:e8:b7:c4:b0:d1:a5:c0:e2:d8:a3:81:f2:86:
e7:93:f6:d4:ee:df:a2:df:a2

Info
00:00:00:1d:d0:f2:81:f8:8b:ff:9a:f7:d5:ef:94:b6:d1:b4:c0:9f:ec:95:e6:8f:e1:87:e8:ca:f0:8b:f6:8b:f6

Nearby Networks
00:00:00:28:d0:f2:9c:f9:8d:e4:82:a0:9a:e1:c3:a4:c1:b5:ea:99:fa:9b:f5:9c:f2:94:fb:d9:e3:98:ba:c8:ad:cb:b9:dc:af:
c7:e5:df:ef:92:ef:92

Schedule
00:00:00:1d:d0:f2:81:e2:8a:ef:8b:fe:92:f7:d5:ef:94:b6:d1:b4:c0:9f:ed:98:f4:91:e2:c0:fa:81:fc:81:fc

energy
00:00:00:1e:d0:f2:97:fa:9f:eb:8e:fc:de:e4:9f:bd:da:bf:cb:94:e6:83:e2:8e:fa:93:fe:9b:b9:83:f8:85:f8:85

Reboot
00:00:00:21:d0:f2:81:f8:8b:ff:9a:f7:d5:ef:94:b6:c4:a1:c3:ac:c3:b7:95:af:d4:f6:92:f7:9b:fa:83:a1:9b:aa:d7:aa:d7

Factory Reset
00:00:00:20:d0:f2:81:f8:8b:ff:9a:f7:d5:ef:94:b6:c4:a1:d2:b7:c3:e1:db:a0:82:e6:83:ef:8e:f7:d5:ef:de:a3:de:a3
```

Script Specifics

Since we were able to find out the algorithm the app uses to encrypt local communication, our script is able to decode the response from the plug. That means when we make a request to see the energy consumption, nearby networks, schedule, or general information regarding the plug, we can understand the response. A hacker could use this information to generalize when a user is home or not and plan more attacks. They could even disconnect plug completely from the network using factory reset and wait for the plug user to resend information over the network to be exploited as well. The script also allows us to spam the plug with on and off commands for a specific duration with a specified interval (ie turn the plug on and off for 10 seconds in intervals of .5 seconds). We also added functionality to scan the network in parallel so an attacker does not even need the ip's of smart plugs on the network. This extends to sending commands to multiple smart plugs at once, so one could toggle all the plugs on a network at the same time, get all their information, turn them all off, etc.

Wrap Up

Overall, we were able to successfully send unverified packets to the tp link plug while on the same network. We were unable to access the cloud service without proper authentication using methods we learned about in class, but this could be a future area of research to find more vulnerabilities. We were able to leverage our ability to send unverified packets locally to send requests such as how much energy the plug is using, what networks are nearby, etc and decrypt their responses after researching online. Finally, we set up a toggle function to quickly turn the plug on and off as frequently as needed which could induce seizures/break some devices.