Securifi Almond 3

Internal / Testing

November 12, 2019 – Version 1.0

Prepared by

Josh Rabinowitz

Abstract:

The Almond 3 is a “smart home Wi-Fi system” made by Securifi. While IoT products like this are growing in popularity and complexity, they are also putting homes and businesses at risk due to the inherent lack of built-in security. As a router capable of port forwarding and controlling multiple low-energy sensor networks, the Almond 3 gives any successful hacker a dangerous amount of power. With that in mind, I decided to assess the security of each of the Almond’s components: web application, mobile application, networks and hardware. Throughout this paper I detail the methodologies used, vulnerabilities I found, and several ideas for further research.

Table of Contents

Introduction

Just as the internet connected people across the globe, the Internet of things (IoT) is an emerging phenomenon that extends this connectivity to a wide variety of embedded devices and everyday objects. Ranging from TV's and thermostats to lightbulbs and refrigerators, these devices can communicate and interact with other IoT-capable devices and provide remote control and monitoring. Essentially, an IoT device is an embedded device with internet connectivity. As these devices are being utilized in both homes and business environments, some projections indicate that we will see as many as 20 billion IoT devices connected to the internet within the next few years.

While these smart devices provide a new level of convenience and functionality, they often provide an inherent security risk as well. With the market in competition over this new phenomenon companies are in a hurry to get their IoT designs into production. These designs also tend to focus on maximizing functionality in order to compete with similar products, and security is often a low priority or an afterthought rather than a core part of the product development cycle. As a result, smart devices often represent the weak point in a network that allow threat actors a way to compromise the network perimeter and pivot to attack other devices. Many IoT consumers are unaware of these inherent security risks. Moreover, many of these devices are designed and marketed specifically to provide home security.

One such product is the Almond 3 by Securifi. The Almond 3 is a "Smart Home Wi-Fi System". Securifi provides the following description on the Almond 3 product page:

> "Meet the Almond 3, our latest innovation that easily and quickly establishes a single, secure Home Wi-Fi System in minutes. To create a Home Wi-Fi Network, simply setup two or more Almond 3 units. It's easy with our innovative easy touchscreen access and Wizard."

Essentially, the Almond 3 is a smart router that is highly customizable and can be controlled remotely via a web-based dashboard or a mobile application. The device can be configured to replace a home router or act as a range extender, and multiple Almond 3 devices can be set up to act as a mesh network and provide robust connectivity throughout large homes or areas.

Overall, the Almond 3 promises to be a fast, airtight home security system that provides the user with fine grained control over their network and devices. Through a thorough security assessment of the Almond 3 and all of its components, I found that this device actually opens up a home network to outside attacks and may ultimately place that fine grained control within the hands of a malicious threat actor.

Scope

With several interconnected components and such a wide range of features and operations, the Almond 3 bears a rather large and fruitful attack surface. Assessing the security of this product involved several focused areas of security including web security, mobile security, network security, hardware hacking and reverse engineering.

The following subsections detail the various components that made up the scope of this assessment.

Web Application

Like most ordinary routers, the Almond 3 comes equipped with a web-based HTTP dashboard for simple, remote administration. By logging in with your Almond user account you can control just about every aspect of your device or network without physical access to the Almond 3. This also means that any attacker who can take control of the web application can ultimately control the Almond 3 and its network remotely. It's also worth mentioning that the Almond 3 allows port forwarding, which means the device can be administered from anywhere in the world with internet access; this makes any web-based vulnerabilities even more critical.

Mobile Application

As an alternative to the web-based dashboard (or for additional remote access), users can download the free application from the Apple[1] or Google Play[2] stores to control the Almond 3 and its network from an iPhone or an Android phone, respectively. Naturally, the addition of a mobile application provides an additional target for hackers. Although there is an Almond application for both Android and iOS, for the sake of time only the Android application was tested during this assessment.

Networks

Between the web and mobile applications, quite a bit of TCP/IP network traffic flows to and from the main Almond 3 access point or range extender. In addition, any additional Almond devices paired together or in a mesh will produce even more traffic. Finally, the addition of any other IoT devices or low-power networks (Bluetooth, Zigbee, Z-wave) will yield yet another layer of network traffic that can potentially be sniffed, captured, spoofed or replayed in some way that allows an attacker to compromise the network or one of its devices.

Hardware

Lastly, of course, is the device itself. To protect the Almond 3 from unauthorized use, the device comes equipped by default with a timed screen lock that can only be opened with a configurable PIN key. If an attacker obtains physical access to the device, however, they can physically take it apart in an attempt to tamper with it or interface with the circuitry. As such, the device and its physical components provide a final target in the event that the device cannot be compromised remotely.

[1] Apple App Store. Almond by Securifi. https://apps.apple.com/us/app/almond-by-securifi/id908025757.
[2] Google Play Store. Almond. https://play.google.com/store/apps/details?id=com.securifi.almondplus&hl=en_US.

WEB

The web application was assessed with a standard web application penetration testing methodology, using Burp Suite[3] and Firefox while referencing the OWASP Top 10 web vulnerabilities as a guideline.

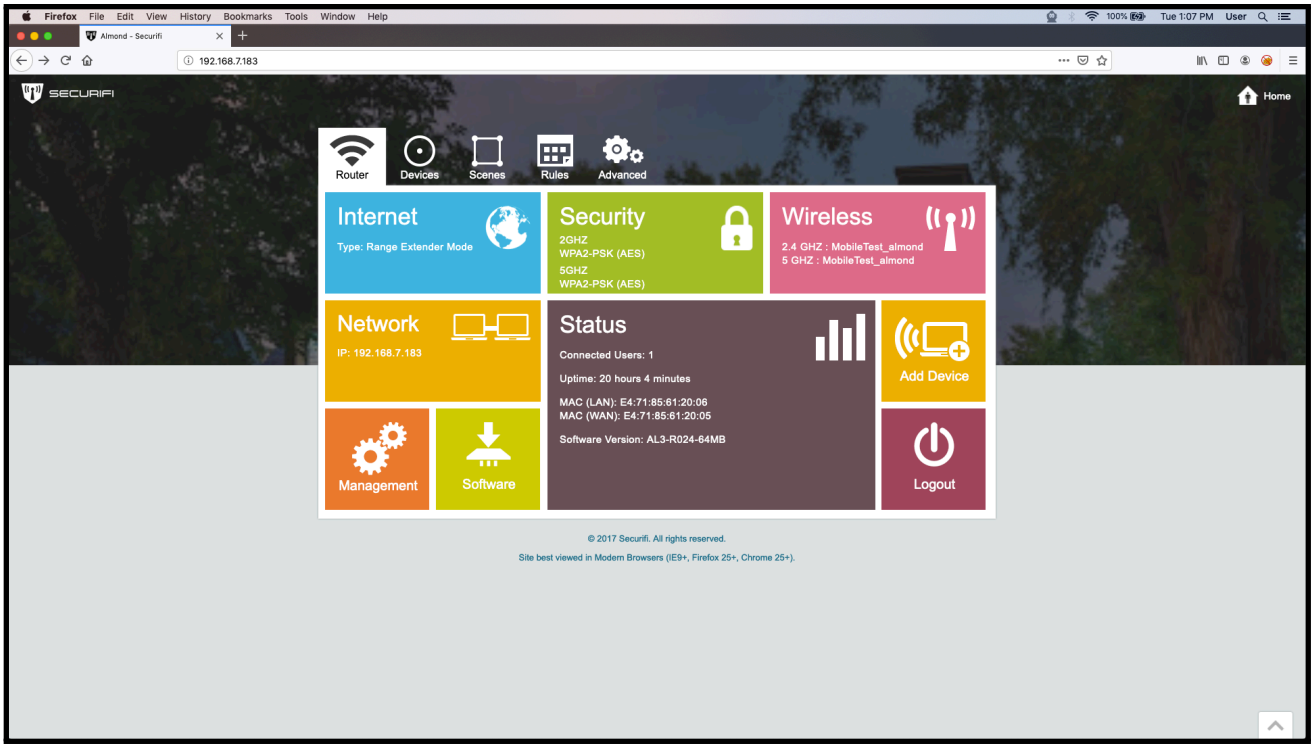

Figure 1: Dashboard Interface

Authentication

As the web application is an administrative dashboard it requires the user to authenticate using their Almond account, so I began the assessment by testing the authentication mechanism. Looking at some of the requests/responses using Burp Suite's proxy tool I noticed immediately that the application leaves all web traffic unencrypted, so I was able to view all of the server's traffic without decrypting. By logging in with the proxy tool listening I captured the following request:

```
GET / HTTP/1.1
Host: 192.168.7.183
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Authorization: Basic YWRtaW46YWRtaW4=
```

Seeing the 'Authorization' HTTP header I recognized this as Basic Authentication, which is trivial to bypass. I then used Burp's "Send to Decoder" feature to decode the value as Base64 and obtain the login credentials.

[3] Portswigger- Burp Suite. https://portswigger.net/burp.

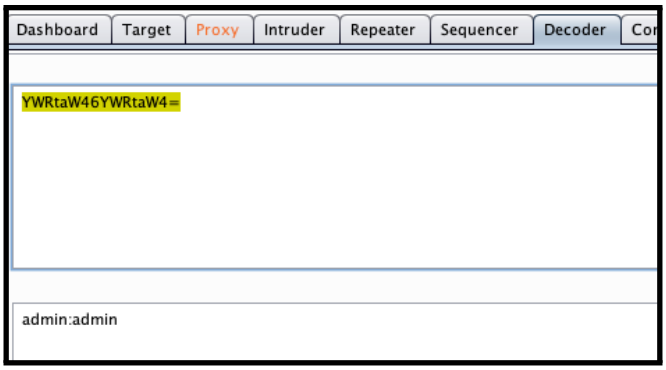

Figure 2: Decrypting Base64 Credentials

Information Disclosure / Hardcoded Secrets in Source Code

Once logged into the dashboard I crawled the application manually while running Burp's web spider to map out the entirety of the site. I found that the primary webpages are '.shtml' files rather than standard '.html'. These are HTML files that use Server Side Includes (SSI), interpreted server-side scripts within the documents (similar to .ASP) that allow the pages to load as fast as possible. By reviewing these included scripts, I was able to deduce much of the server-side functionality. Within these scripts, the server communicates with the master Almond device in real time by querying CGI endpoints and 'goform's. Several of these endpoint queries can be replayed by an attacker to obtain sensitive information from the device.

Sifting through the disclosed source code, I found a number of hardcoded secrets. Among some of these were private IP addresses, MAC addresses, Wi-Fi SSID's and passwords, file path disclosures, developer comments, and emails. I also came across an Oath URI for logging into a Nest account, which contained a specific client ID as a parameter. Based on the surrounding JavaScript, I assume this is used to link the Almond with any Nest products resident on the local network. I tried opening this link in a browser, but was redirected to an empty login page for Nest.

Later in the assessment, after obtaining the filesystem from the device (see HARDWARE section), I was able to directly navigate the web server's root directory and identify web resources that I had missed earlier. One of these was a page called 'version'; viewing this page in the browser opens a text file containing the following string, which identifies the firmware version running on the device: `AL3-R024-64MB`

Frameable Response (Potential Clickjacking)

I found that in several instances the browser renders the page within a frame, and that the application does not set the appropriate X-Frame-Options or Content-Security-Policy HTTP headers in order to prevent framing attacks. As these pages can be loaded within an iframe, this may enable a clickjacking attack in which a malicious page transparently overlays the 'victim' page and deceives the user into following an attacker-controlled link.

To test if the dashboard is vulnerable, I crafted the following proof-of-concept .html document and fed it into the browser:

```
<html>
   <head>
     <title>Clickjacking test:</title>
   </head>
   <body>
     <p>Website is vulnerable to clickjacking!</p>
     <iframe src="http://192.168.7.183/home.shtml" width="500" height="500"></iframe>
```

```
    </body>
</html>
```

When the page loaded I observed this response before being quickly redirected back to the 'real' /home.shtml page:

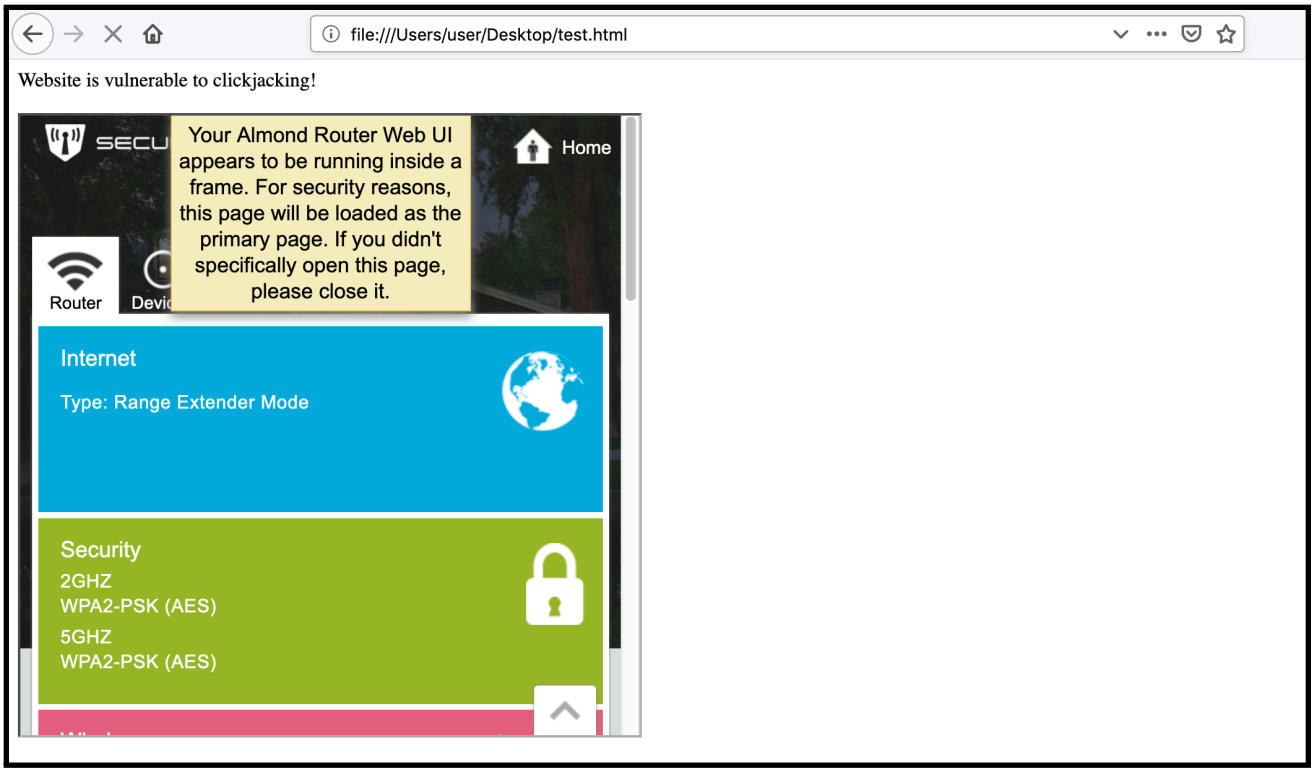

Figure 3: Clickjacking Test

The application appears to have caught my attempt. With a bit more static analysis I was able to locate the 'frame-busting' code that blocked my framing attack:

```
if(top!=self){
    top.location.replace(document.location);
    alert("Your Almond Router Web UI appears to be running inside a frame. For security
    reasons, this page will be loaded as the primary page. If you didn't specifically open
    this page, please close it.");
}
```

Fortunately, it is possible to "bust frame-busting"; this very common client-side protection function can be bypassed in several ways, some of which are browser-specific, even by a relatively unskilled attacker. Some of these methods include double framing, disabling JavaScript, and using the 'onBeforeUnload' event. As such, I decided to skip this frame-busting workaround for the time being and leave it as an exercise for further research.

Exposed/Unencrypted WebSocket

In addition to the standard HTTP traffic on port 80, Burp Suite's proxy caught a GET request coming from port 7681 of the web server. This request contained the login credentials in plain text, which would provide any attacker positioned on the local network with easy access to the dashboard. This request piqued my interest, as I had already authenticated

using Basic Authentication as mentioned previously.

```
GET /admin:admin HTTP/1.1
Host: 192.168.7.183:7681
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://192.168.7.183
Sec-WebSocket-Key: ja9Iifa+v0vsz4IclHfqDQ==
Connection: keep-alive, Upgrade
Cookie: selectedTab=%23tabs-1
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
```

Listing 1: WebSocket Request

```
HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: JiOsYbATM51ER7l3bkyyTS4ROMY=
```

Listing 2: WebSocket Response

Upon closer look I found that this request was a protocol switch from HTTP to WebSocket. WebSocket is used to establish a full duplex connection between the browser and the server and in this case appeared to be proxying system commands between the web dashboard and the device. Similar to HTTP, WebSocket is a plaintext protocol and should be supplemented with TLS for secure connections. In this case the traffic is unencrypted, and I was able to view the WebSocket messages under Burp Suite's 'WebSockets history' tab. The listener port in this instance was on 8080, which a cursory Nmap scan recognized as an http proxy.

Going through the WebSocket messages I noticed that all messages to the server take the following form:

```
{ "MobileInternalIndex" :[#], " CommandType" : " [Command]" }
```

'Command' is one of the following:

- RuleList
- ClientList
- DeviceList
- GetIndexList
- VariableList
- DynamicSceneList
- UpdateAlmondMode

The WebSocket server responds to these commands as expected, returning the results as JSON. 'DeviceList' and 'ClientList' seem to be the most sensitive commands as far as information disclosure; some items of interest among the responses were user email addresses, client IP and MAC addresses, device manufacturers, room/scene information, and Almond sensor/device data. This information could aid a hacker in furthering their attack on the Almond network and its devices.

Although I did not go any further in probing the WebSocket, this would be a good area for further research. It would be quite trivial to build a custom fuzzer, using a small Python script for instance, for testing edge cases (e.g. mutating known commands, trying new ones, attempting to overflow a buffer on the listening side, etc.). Because I was able to sniff the secret WebSocket key, I could now imitate/spoof the web server and, skipping over the browser, communicate directly with the WebSocket server.

File Upload Functionality

One of the dashboard's most powerful capabilities is manually updating the Almond's firmware remotely using a file upload form. Seeing the potential for a very critical 'DoS' (Denial of Service) vulnerability, I attempted to upload a random Word document in place of a valid firmware image. However, the dashboard has a check in place and it caught my incorrect filetype based on the 'Magic Number', a short sequence of bytes at the beginning of a file that indicates the file's type.

Figure 4: Attempting to upload invalid firmware image

If the only check that's in place is a Magic Number check, it would be easy to circumvent this by downloading the latest Almond 3 firmware from the Securifi website,[4] dumping its contents as hexadecimal, and obtaining the valid Magic Number from the first few bytes. By prepending these bytes to the beginning of any file, I could potentially trick the form into uploading any random file as if it were a valid firmware image.

Figure 5: Almond Firmware Images

The risk here is that an attacker could abuse this capability in order to re-flash the device's nonvolatile memory with garbage data, essentially bricking the device. Beyond a DoS, the attacker could upload a valid firmware image that has been modified (to include some form of backdoor or malware, for instance). As testing this issue further would potentially render the device inoperable, I decided to leave this as another area for further research.

Remote Code Execution

Exploring the advanced configuration page I found a 'Console' feature, which allows the user to execute system commands on the Almond 3 remotely- a powerful but dangerous feature to include with a network security system. By executing commands and observing the resulting web traffic, I determined that the command strings are ultimately sent within the "command" parameter of a POST request to /cgi-bin/adm.cgi. For example, the following request is used to execute the 'id' command against the Almond OS:

[4] *How to Update your Almond?* https://www.securifi.com/software-update.

```
POST /cgi-bin/adm.cgi HTTP/1.1
Host: 192.168.7.183
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 48
Authorization: Basic YWRtaW46YWRtaW4=
Connection: close
Referer: http://192.168.7.183/advanced/system_command.shtml
Cookie: selectedTab=%23tabs-3
Upgrade-Insecure-Requests: 1

page=sysCMD&command=id&SystemCommandSubmit=Apply
```

This allows remote authenticated users to interface directly with the underlying OS and execute arbitrary shell commands. By crafting a POST request containing the captured Authorization header, an attacker positioned on the network could easily execute system commands within the context of the web application's privileges.

MOBILE

In addition to the web application Securifi developed mobile applications for Android and iOS. In this section I assess the security of the Android application, downloaded from the Google Play store.

Profiling

I began my assessment by mapping the application's attack surface using Drozer[5] , a convenient fuzzing and exploitation framework for Android. I first gathered some basic information about the Almond APK, including the package information shown below.

```
Package: com.securifi.almondplus
  Application Label: Almond
  Process Name: com.securifi.almondplus
  Version: 7.77.3
  Data Directory: /data/user/0/com.securifi.almondplus
  APK Path: /data/app/com.securifi.almondplus-1/base.apk
  UID: 10094
  GID: [3003]
  Shared Libraries: null
  Shared User ID: null
  Uses Permissions:
  - android.permission.ACCESS_NETWORK_STATE
  - android.permission.ACCESS_WIFI_STATE
  - android.permission.CALL_PHONE
  - android.permission.CAMERA
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.INTERNET
  - com.google.android.c2dm.permission.RECEIVE
  - android.permission.VIBRATE
  - android.permission.RECEIVE_BOOT_COMPLETED
  - android.permission.RECORD_AUDIO
  - android.permission.READ_PHONE_STATE
  - android.permission.GET_ACCOUNTS
  - android.permission.WAKE_LOCK
  - com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE
  Defines Permissions:
  - None
```

I also took a look at the application's manifest file, and noticed that the application supports older versions of Android SDK, with the minimum version being 16. Version 16 corresponds to Android 4.1 "Jelly Bean" which has been unmaintained for several years. Versions of Android which no longer receive updates from Google remain vulnerable to many security flaws, which could leave the Almond system open to compromise through the underlying platform. This is more dangerous given the number of permissions the application requests.

Data Storage

After profiling the application, I continued my assessment by testing the application's IPC mechanism. As this was effectively a black-box test, I realized I'd be better off performing static analysis first, then returning to IPC once I knew how the exported components handled my input. To that end, I first checked how the application stores its data. What I was most interested in was any sensitive information the application might store on the device, which an attacker could use to get into the application and take control of the Almond network. This is where my information gathering efforts came in handy; having obtained the package information earlier I knew where to find the application's data directory. The mobile device I used for testing did not allow running the adb daemon as root, so I had to use my

[5] MWR Labs - Drozer. https://labs.mwrinfosecurity.com/tools/drozer/.

root permissions within the adb shell to copy the application's data directory to a world-readable location as shown. Alternatively, I could have used a modified version of 'adbd' that can run as root in production builds, but that would've taken a bit more time.

```
admins-MacBook-Pro:~ user$ adb shell su
root@R1_HD:/ # mkdir /sdcard/data
mkdir /sdcard/data
root@R1_HD:/ # cp -r /data/user/0/com.securifi.almondplus /sdcard/data
cp -r /data/user/0/com.securifi.almondplus /sdcard/data
cp: lib: Function not implemented
1|root@R1_HD:/ # exit
exit
admins-MacBook-Pro:~ user$ adb pull /sdcard/data
/sdcard/data/: 28 files pulled. 4.3 MB/s (783255 bytes in 0.175s)
admins-MacBook-Pro:~ user$ tree -L 3 ./data
./data
└── com.securifi.almondplus
    ├── app_webview
    │   ├── Web\ Data
    │   ├── Web\ Data-journal
    │   ├── metrics_guid
    │   └── webview_data.lock
    ├── cache
    │   └── volley
    ├── code_cache
    │   └── com.android.opengl.shaders_cache
    ├── databases
    │   ├── almond_database
    │   ├── almond_database-shm
    │   ├── almond_database-wal
    │   ├── google_app_measurement.db
    │   ├── google_app_measurement.db-journal
    │   ├── google_app_measurement_local.db
    │   └── google_app_measurement_local.db-journal
    ├── files
    │   ├── Mint-lastsavedfile
    │   ├── MintSavedData-1-1562612952465.json
    │   ├── crashCounter
    │   ├── google_app_measurement.db
    │   └── lastCrashID
    ├── no_backup
    │   ├── com.google.InstanceId.properties
    │   └── com.google.android.gms.appid-no-backup
    ├── program_cache
    └── shared_prefs
        ├── Mint.xml
        ├── REMOTESETTINGSSETTINGS.xml
        ├── UIDPREFERENCES.xml
        ├── WebViewChromiumPrefs.xml
```

```
        ├── com.google.android.gms.appid.xml
        ├── com.google.android.gms.measurement.prefs.xml
        └── com.securifi.almondplus_preferences.xml

9 directories, 27 files
```

Here I provide a brief summary of what I found within these directories:

- The WebView directory contains autofill, server, and credit card information (I found later, during Static Analysis , that the mobile application uses Stripe[6] to process payments for automated vulnerability scans, another one of Almond's many features).
- The caches contain information about recent connections and client devices, but nothing sensitive.
- The database directory contains a database with tables concerning android metadata, recent activity, sync points, notifications, and room information
- The "files" directory contains crash information, as well as JSON data tracking the device using the mobile application, including physical coordinates
- The preferences directory contains a handful of .xml files, one of which contains temporary passwords stored in base64, as well as plaintext passwords for the web dashboard and the PIN to unlock the touchscreen, in addition to other sensitive information such as emails and IP addresses

Overall, I found a great deal of sensitive information stored in plaintext, or in some cases in base64 encoding. If an Android device has been rooted, an attacker with physical access can use the application's data storage to obtain critical information and launch further attacks on the network.

Static Analysis

Having profiled the application and obtained the package information, I knew the base APK file was located in "/data/app/com.securifi.almondplus-1/". Using the same steps as above, I pulled from the device the directory containing the APK file, 'base.apk'. I began reverse engineering using apktool[7] , an all-in-one tool that decompresses the apk, uses Backsmali on the dex file for disassembly, and retrieves the manifest as an XML file. This yielded the following directory:

```
├── AndroidManifest.xml
├── apktool.yml
├── assets
├── original
├── res
├── smali
└── unknown
```

Aside from the digital signature file 'META-INF' under original, I was most interested in the 'smali' directory, the structure of which is pictured below.

```
├── android
|   ├── arch
|   |   ├── a
|   |   ├── lifecycle
|   |   └── persistence
|   └── support
|       ├── annotation
```

[6] Stripe. https://stripe.com.
[7] ibotpeaches. Apktool - Github. https://ibotpeaches.github.io/Apktool/.

```
|       ├── asynclayoutinflater
|       ├── compat
|       ├── constraint
|       ├── coordinatorlayout
|       ├── coreui
|       ├── coreutils
|       ├── customview
|       ├── design
|       ├── drawerlayout
|       ├── fragment
|       ├── graphics
|       ├── loader
|       ├── mediacompat
|       ├── multidex
|       ├── slidingpanelayout
|       ├── swiperefreshlayout
|       ├── transition
|       ├── v4
|       └── v7
├── androidx
|   ├── core
|   |   └── graphics
|   ├── media
|   |   ├── AudioAttributesCompatParcelizer.smali
|   |   ├── AudioAttributesImplApi21Parcelizer.smali
|   |   └── AudioAttributesImplBaseParcelizer.smali
|   └── versionedparcelable
|       ├── CustomVersionedParcelable.smali
|       ├── ParcelImpl.smali
|       ├── a.smali
|       ├── b.smali
|       └── (…)
├── antistatic
|   └── spinnerwheel
|       ├── AbstractWheel$SavedState.smali
|       ├── AbstractWheel.smali
|       ├── AbstractWheelView.smali
|       ├── R$attr.smali
|       ├── R$style.smali
|       ├── R$styleable.smali
|       ├── R.smali
|       ├── WheelHorizontalView.smali
|       ├── WheelVerticalView.smali
|       ├── a
|       ├── a.smali
|       ├── b.smali
|       └── (…)
```

```
├── com
│   ├── a
│   │   └── a
│   ├── android
│   │   └── volley
│   ├── b
│   │   ├── a
│   │   ├── b
│   │   └── c
│   ├── c
│   │   └── a
│   ├── decoder
│   │   └── util
│   ├── encoder
│   │   └── util
│   ├── google
│   │   ├── android
│   │   ├── firebase
│   │   └── zxing
│   ├── journeyapps
│   │   └── barcodescanner
│   ├── larswerkman
│   │   └── holocolorpicker
│   ├── securifi
│   │   ├── a
│   │   └── almondplus
│   ├── splunk
│   │   └── mint
│   ├── squareup
│   │   └── timessquare
│   ├── stripe
│   │   └── android
│   └── tutk
│       └── IOTC
└── org
    └── bytedeco
        ├── javacpp
        └── javacv
```

As evidenced by the directory structure, I found several third-party software components associated with Google, Stripe, Mint and more. However, at this point I was most interested in Securifi's proprietary code (mostly under 'almondplus'), which is most likely to retain vulnerabilities. Apktool produces Smali, a more readable version of Dalvik bytecode (.dex files), which is most accurate but not easy to quickly understand. In order to understand the code at a higher level I ended up turning to 'dex2jar'[8] , which decompiles the application into Java; this is less accurate but lends itself to quick reverse engineering. To view the resulting class files I used a GUI utility called 'JD-GUI'[9] .

[8] pxb1988. dex2jar –Github. https://github.com/pxb1988/dex2jar.
[9] jd-gui –Github. https://github.com/java-decompiler/jd-gui.

When logging into the mobile application, the user is given the option of connecting locally to the Almond device or connecting through the Internet via the Securifi cloud server. Looking at the Java classes I found the code responsible for these password checks, and I also found the 'receiver' classes that handle the various updates and messages to and from the Almond device; I kept these handy for when I tested the IPC components later (see IPC section). As the code was still slightly obfuscated, I was not able to immediately trace where incoming remote traffic is 'handled'. In the 'sdk' package under the 'almondplus' directory, I found a class that appears to be responsible for encrypted communications with the Securifi cloud server (which, as mentioned earlier, is out of scope for this assessment). This piece of code contains a hardcoded certificate, along with the associated domain name(s) and what I can safely assume is the key length (1028 bits).

Figure 6: Hardcoded Certificate Information

Although I did not use the certificate to try and connect to Securifi's server, I used 'openssl' to check the encoded information, including the signature algorithm and the associated CA (certificate authority):

```
admins-MacBook-Pro:tmp user$ openssl x509 -in cert.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 144470 (0x23456)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, O=GeoTrust Inc., CN=GeoTrust Global CA
        Validity
            Not Before: May 21 04:00:00 2002 GMT
            Not After : May 21 04:00:00 2022 GMT
        Subject: C=US, O=GeoTrust Inc., CN=GeoTrust Global CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:da:cc:18:63:30:fd:f4:17:23:1a:56:7e:5b:df:
                    3c:6c:38:e4:71:b7:78:91:d4:bc:a1:d8:4c:f8:a8:
                    43:b6:03:e9:4d:21:07:08:88:da:58:2f:66:39:29:
                    bd:05:78:8b:9d:38:e8:05:b7:6a:7e:71:a4:e6:c4:
                    60:a6:b0:ef:80:e4:89:28:0f:9e:25:d6:ed:83:f3:
                    ad:a6:91:c7:98:c9:42:18:35:14:9d:ad:98:46:92:
                    2e:4f:ca:f1:87:43:c1:16:95:57:2d:50:ef:89:2d:
                    80:7a:57:ad:f2:ee:5f:6b:d2:00:8d:b9:14:f8:14:
                    15:35:d9:c0:46:a3:7b:72:c8:91:bf:c9:55:2b:cd:
```

```
                d0:97:3e:9c:26:64:cc:df:ce:83:19:71:ca:4e:e6:
                d4:d5:7b:a9:19:cd:55:de:c8:ec:d2:5e:38:53:e5:
                5c:4f:8c:2d:fe:50:23:36:fc:66:e6:cb:8e:a4:39:
                19:00:b7:95:02:39:91:0b:0e:fe:38:2e:d1:1d:05:
                9a:f6:4d:3e:6f:0f:07:1d:af:2c:1e:8f:60:39:e2:
                fa:36:53:13:39:d4:5e:26:2b:db:3d:a8:14:bd:32:
                eb:18:03:28:52:04:71:e5:ab:33:3d:e1:38:bb:07:
                36:84:62:9c:79:ea:16:30:f4:5f:c0:2b:e8:71:6b:
                e4:f9
            Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                C0:7A:98:68:8D:89:FB:AB:05:64:0C:11:7D:AA:7D:65:B8:CA:CC:4E
            X509v3 Authority Key Identifier:
                keyid:C0:7A:98:68:8D:89:FB:AB:05:64:0C:11:7D:AA:7D:65:B8:CA:CC:4E

    Signature Algorithm: sha1WithRSAEncryption
         35:e3:29:6a:e5:2f:5d:54:8e:29:50:94:9f:99:1a:14:e4:8f:
         78:2a:62:94:a2:27:67:9e:d0:cf:1a:5e:47:e9:c1:b2:a4:cf:
         dd:41:1a:05:4e:9b:4b:ee:4a:6f:55:52:b3:24:a1:37:0a:eb:
         64:76:2a:2e:2c:f3:fd:3b:75:90:bf:fa:71:d8:c7:3d:37:d2:
         b5:05:95:62:b9:a6:de:89:3d:36:7b:38:77:48:97:ac:a6:20:
         8f:2e:a6:c9:0c:c2:b2:99:45:00:c7:ce:11:51:22:22:e0:a5:
         ea:b6:15:48:09:64:ea:5e:4f:74:f7:05:3e:c7:8a:52:0c:db:
         15:b4:bd:6d:9b:e5:c6:b1:54:68:a9:e3:69:90:b6:9a:a5:0f:
         b8:b9:3f:20:7d:ae:4a:b5:b8:9c:e4:1d:b6:ab:e6:94:a5:c1:
         c7:83:ad:db:f5:27:87:0e:04:6c:d5:ff:dd:a0:5d:ed:87:52:
         b7:2b:15:02:ae:39:a6:6a:74:e9:da:c4:e7:bc:4d:34:1e:a9:
         5c:4d:33:5f:92:09:2f:88:66:5d:77:97:c7:1d:76:13:a9:d5:
         e5:f1:16:09:11:35:d5:ac:db:24:71:70:2c:98:56:0b:d9:17:
         b4:d1:e3:51:2b:5e:75:e8:d5:d0:dc:4f:34:ed:c2:05:66:80:
         a1:cb:e6:33
```

In the third-party directories I found several hardcoded URI's and API keys, some of which appear to be in code that crafts POST requests to the corresponding e-commerce platforms. Among some of these were:

- "https://api.stripe.com"
- "https://m.stripe.com/4" (visiting this link returns a session ID: *7c591f70-d2d1-44fe-b134-3410b389784a*)
- "https://q.stripe.com"
- "splkmobile.com"
- "http://5824d841.api.splkmobile.com"

Within the Securifi directory I found several more hardcoded links:

- "https://firmware.securifi.com/CA1/version"
- "https://firmware.securifi.com/A3S/version"
- "https://firmware.securifi.com/A3S_BETA/version"
- "https://firmware.securifi.com/AL3_64MB/version"
- "https://firmware.securifi.com/AL3_64MB_BETA/version"
- "https://firmware.securifi.com/AL3_BETA/version"
- "mobile.securifi.com"
- "https://utils.securifi.com" (visiting this link returns *"hello world"*)
- "https://utils.securifi.com/RecentActivity"
- "https://sendlogs.securifi.com:1027/getUrls"
- "https://utils.securifi.com/ResendActivationLink"
- "https://utils.securifi.com/ResetPassword"
- "https://utils.securifi.com/almond/changeMode"
- "https://utils.securifi.com/almonds"
- "cloud.securifi.com"
- "https://sitemonitoringdev.securifi.com:8082"
- "https://utils.securifi.com/SiteMonitoring"
- "https://home.nest.com/login/oauth2?client_-id=cfe09fe9-3805-4a9f-8c5b-90128f650584&state=STATE"

Dynamic Analysis / Communication

In order to monitor some of the external communication in more detail I turned to dynamic analysis, proxying the application's traffic through Burp Suite and using 'logcat' to monitor system messages within Android. As the application does not utilize a network security policy or certificate pinning, it was sufficient to install the Burp Suite certificate into the Android OS trusted key-store. When I logged into the application I observed a strange POST request, which I quickly realized was compressed with gzip.

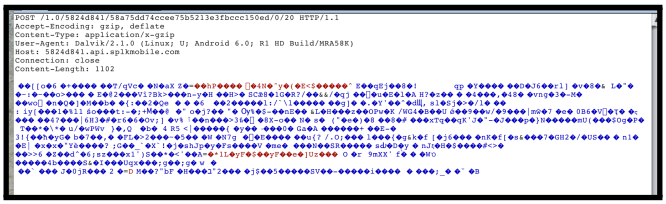

Figure 7: Mobile Traffic

Decoding as gzip (and 'prettifying' the resulting JSON data) revealed the following updates being sent repeatedly in real time:

{
"sdkVersion":"5.2.1", "apiKey":"5824d841", "platform":"Android", "device":"BLU R1 HD", "osVersion":"6.0", "locale":"US", "uuid":"58a75dd74ccee75b5213e3fbccc150ed", "userIdentifier":"NA", "appEnvironment":"Staging", "batteryLevel":100, "carrier":"NA", "remoteIP":"{%#@@#%}", "appVersionCode":"110", "appVersionName":"7.79", "packageName":"com.securifi.almondplus", "connection":"WIFI", "state":"CONNECTED", "currentView":"com.securifi.almondplus.Al
"screenOrientation":"Portrait", "msFromStart":89, "session_id":"6261eb3c-6072-486a-9340-1246598b9f79", "extraData":{}, "transactions":[], "location":{
"longitude":"NA", "latitude":"NA", "timestamp":"NA" }, "current":"com.securifi.almondplus.AlmondPlusActivity", "previous":"NA", "domainLookupTime":"NA", "domProcessingTime":"NA", "serverTime":"NA", "host":"NA", "loadTime":"NA", "elapsedTime":"6151" }{
^1^view^1568056092 }

When the request above is sent, 'https://5824d841.api.splkmobile.com' sends the following response indicating the request has been successfully fulfilled and there is no content to return:

```
HTTP/1.1 204 No Content
content-length: 0
date: Wed, 11 Sep 2019 19:21:52 GMT
server: Cowboy
Connection: Close
```

These real-time updates are sent to a server belonging to Splunk's 'Mobile Intelligence' unit, presumably for metrics/analytics. While the application sends information to Splunk, it also queries a server at 'https://utils.securifi.com' for recent activity, which comes from the logs on the Almond device. These queries include a Bearer token for authorization. Aside from the token, I found nothing interesting in the observed web traffic.

Before moving onto I decided to check for any communication protocols besides web traffic. To create a makeshift network proxy I used the following command, which forwarded all network traffic through Wireshark:

```
nc localhost 4444 | sudo wireshark -k -S -i en0
```

Aside from the web traffic I found DNS queries for 'utils.securifi.com' and 'firmware.securifi.com'; the latter is likely used

to check periodically for available software updates.

IPC

While I was able to obtain the manifest file, Drozer is capable of parsing it automatically to create a quick summary of the attack surface.

```
Attack Surface:
  4 activities exported
  5 broadcast receivers exported
  0 content providers exported
  1 services exported
```

The application exports several components with "null" permissions, which means they are potentially vulnerable to an attacker. Using the activity manager in adb (Android Debug Bridge) I fuzzed each of these exported components manually to find any interesting or unexpected behavior within the application's IPC mechanisms. The exported service is Google's Firebase Messaging Service. Although this appeared vulnerable at first, I found the following notice in the Firebase documentation:

> FirebaseInstanceIdService performs security checks at runtime, no need for explicit permissions despite exported="true"

Moving on, I looked at which broadcast receivers are exported:

```
Package: com.securifi.almondplus
  com.securifi.almondplus.widgets.MultipleActionWidgets
    Permission: null
  com.securifi.almondplus.widgets.AlmondAppWidget
    Permission: null
  com.securifi.almondplus.cloud.UpdateReceiver
    Permission: null
  com.google.firebase.iid.FirebaseInstanceIdReceiver
    Permission: com.google.android.c2dm.permission.SEND
  com.google.android.gms.measurement.AppMeasurementInstallReferrerReceiver
    Permission: android.permission.INSTALL_PACKAGES
```

The first three do not require permissions and therefore may be vulnerable. In order to try and exploit these receivers, I looked at the decompiled Java to determine what valid intents would look like (action, extras, etc.) and how they're handled in the corresponding 'onReceive()' functions. The first two read in 'extras' (parameters that are passed along in the intent), while the third only checks the action. Based on the receiver functions, there's nothing malicious that can be done. Aside from triggering connectivity updates in logcat, I was ultimately not able to do anything with these broadcast receivers. Lastly, I checked the four exported activities and determined that none are vulnerable.

NETWORK

Zigbee

To test the built-in Zigbee server I used an ApiMote V4[10] for sniffing, injection, and analysis of Zigbee packets. The board comes pre-flashed with KillerBee[11] firmware, an open source framework for exploring and exploitation of Zigbee and IEEE 802.15.4 networks. For a test sensor I used a GE Link A19 smart LED bulb, along with a Wink hub, by Quirky, to pair the bulb with the Almond.

I began by using zbstumbler to hop through the channels and determine which is used by the Almond for communicating with sensors. I found that the smart bulb and the Wink hub paired over channel 26, while channel 24 hosts the communications between the smart bulb and the Almond itself.

Having found the correct channel I used zbdump and zbwireshark to sniff the Zigbee and IEEE 802.15.4 traffic while pairing the sensor as well as sending it commands such as on, off and dim. I was able to capture the association request with the PAN ID as well as the EPID of the controller (the Almond), but the payloads within the Zigbee packets are encrypted. Decrypting and analyzing the traffic requires capturing the Transport Key, but unfortunately some of the captured packets were incomplete, resulting in bad frame checksums. A debug message in the console indicated "clearing overflow", which could mean the computer is fetching packets too slowly, there is a proximity issue, or the Almond is using its own proprietary Zigbee stack.

I attempted to communicate with the Almond by crafting packets manually using zbscapy. I wrote the following script to send a beacon request, but this resulted in the same debug message:

```
dev = kbutils.devlist()[0][0]
brpkt = Dot15d4()/Dot15d4Cmd(cmd_id='BeaconReq',dest_addr=0x0000)
kb_send = KillerBee(device=dev)
kb_send.set_channel(24)
dp = kbsrp1(brpkt,iface=kb_send,timeout=1)
if dp: print dp.show()
```

After several unsuccessful attempts I then tried to flood the Almond with association requests using the PAN ID obtained, but was unable to crash the Zigbee server. Lastly, I attempted to replay some of the captured packets, but was unsuccessful.

Additional Networks

While the Almond is advertised as compatible with Z-Wave and Bluetooth, Z-Wave functionality requires the use of a proprietary USB Dongle from Securifi, which must be purchased separate from the Almond itself and is currently sold out. As for Bluetooth, there does not seem to be any compatibility as advertised. The other technologies Almond 3 works with out-of-the-box are Phillips Hue networks and Google's Nest technology. Unfortunately, I did not have sensors/hubs handy to test these, and left them for future research.

[10] riverloopsec. apimote –Github. https://github.com/riverloopsec/apimote.
[11] riverloopsec. killerbee –Github. https://github.com/riverloopsec/killerbee.

HARDWARE

The primary method for interacting with the Almond 3 is the device's LED touchscreen GUI. Some features are only accessible from the device itself. Physical access is protected by a lock screen, which is unlocked using a 4-digit PIN:

Figure 8: Lock Screen

Figure 9: PIN Entry Screen

If an attacker can gain access to the device there are multiple ways to bypass this via the hardware, as Securifi did not utilize any sort of hardware obfuscation or protection. Teardown simply involves unscrewing the two outer screws on the bottom of the device and removing the back of the plastic encasing (I also detached the Piezoelectric buzzer to avoid pulling out the leads).

Component Identification

I began the hardware assessment by identifying the PCB's primary components. A USB Microscope was used to inspect the IC's chip numbers up close, and all datasheets[12,13] were readily available and found through quick searches.

[12] Micron. N25Q512A Datasheet. https://www.micron.com/products/nor-flash/serial-nor-flash.
[13] MediaTek. MT7621 Datasheet. http://www.t-firefly.com/download/FireWRT/hardware/MT7621.pdf.

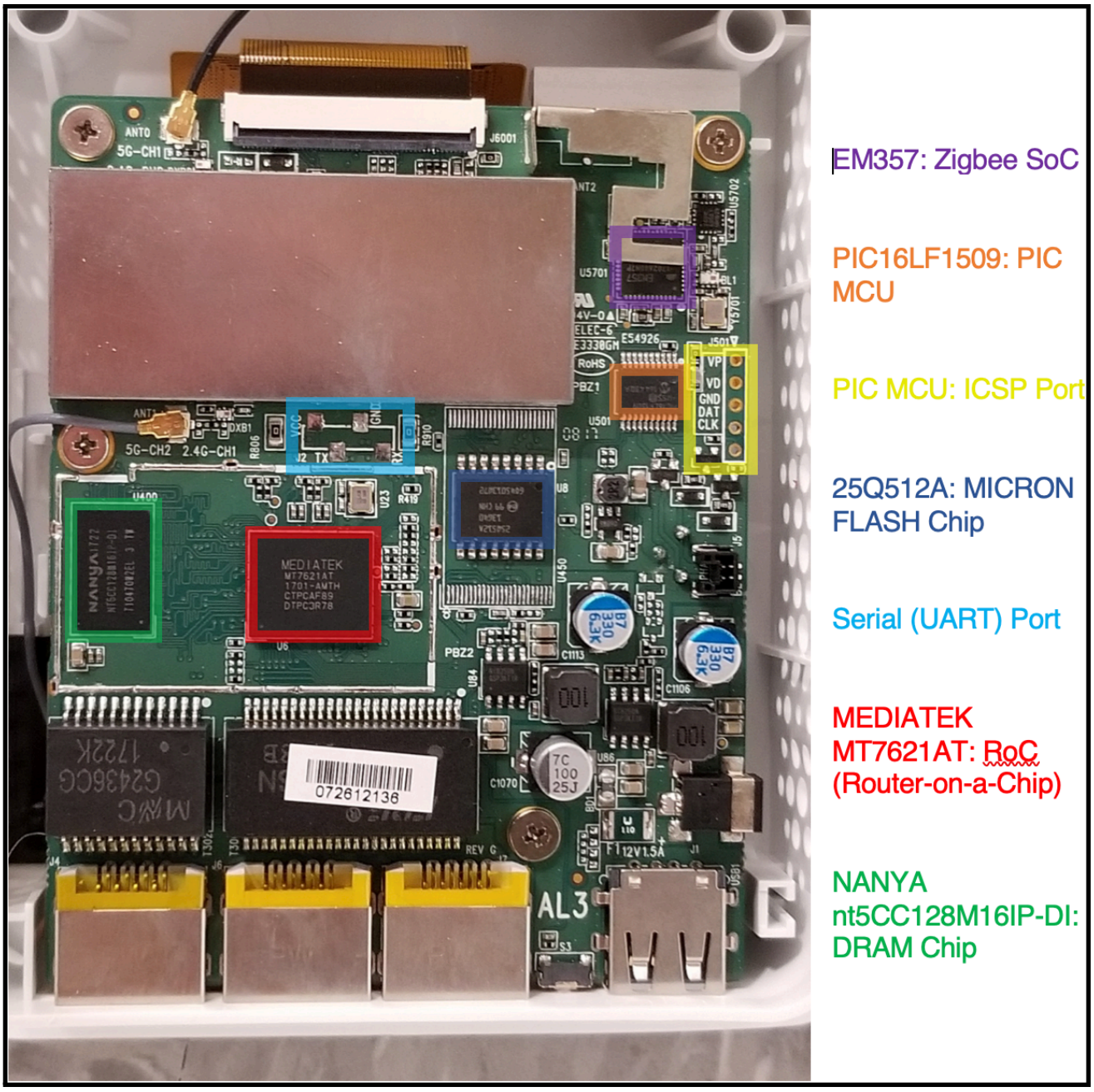

Figure 10: Almond 3 Hardware Components

PIC Microcontroller

The board has a 5-pin ICSP header connecting to the PIC microcontroller unit. To interface with the microcontroller I used an MPLAB ICD 3, a dedicated in-circuit debugger by Microchip, and MPLAB X IDE v5.20, Microchip's proprietary freeware used for developing embedded applications for PIC.

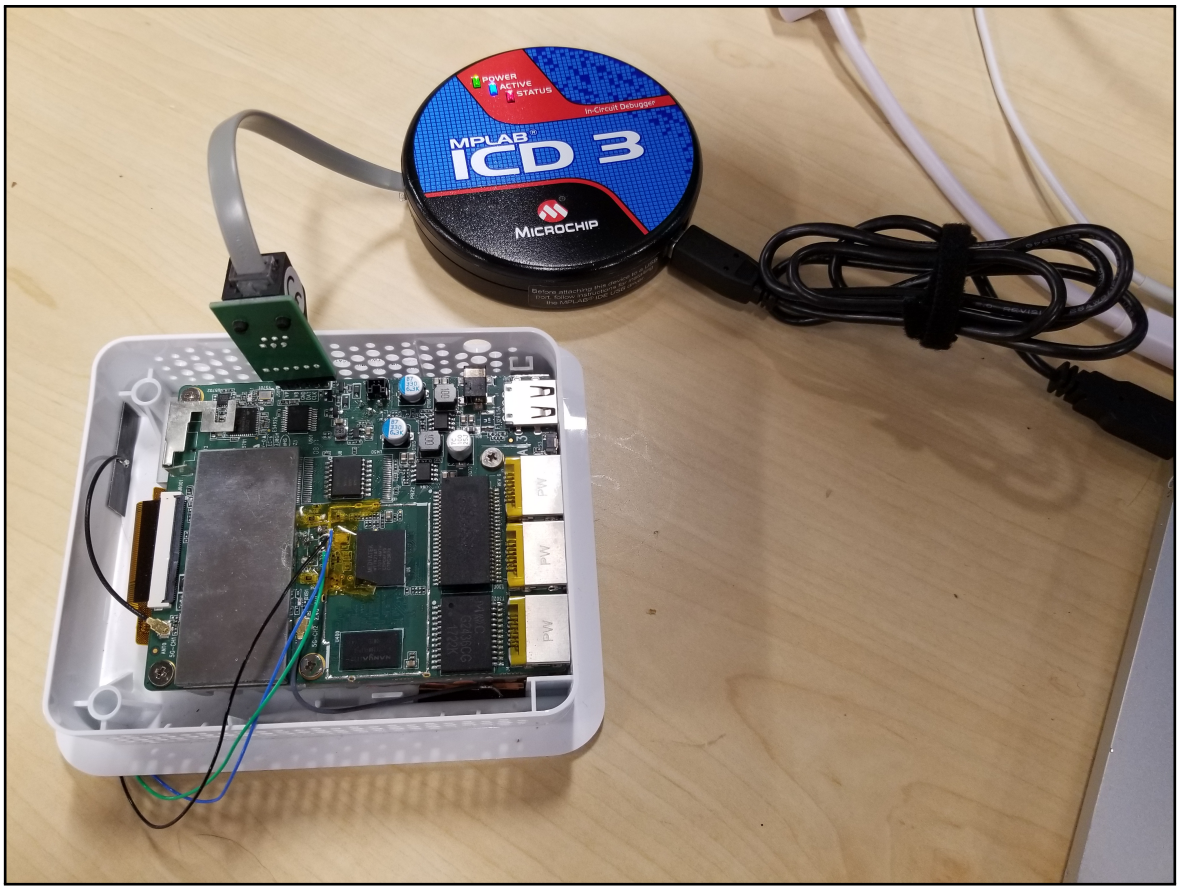

Figure 11: Interfacing with PIC MCU

Figure 12: ICSP Pinout

Reading the chip's contents resulted in a file with the following format:

```
:020000040000FA
:100000008831F329FF3FFF3F70082000A8007108E6
:10001000A7007208A6007308A5007408A4007B0856
:10002000A3007C08A2007D08A1008031911D5D2DF8
:1000300091112400151F212811082000BA00240066
(···)
```

Recognizing this as Intel 'flavored' hexadecimal I imported the file into Ghidra, which correctly interpreted and displayed the assembly with 'PIC16F' selected as the language. Decompiling revealed a 'Reset' function as well as a large interrupt handling function, which appears to dispatch different subroutines based on the contents of the Peripheral Interface and SSP (Synchronous Serial Port) related registers. Although I ran short on time reversing these functions to find vulnerabilities, it's worth noting that Securifi could have made use of PIC's configuration fuses, effectively preventing anyone from reading out the contents of the Microcontroller.

UART

The board's main IC is the MediaTek router-on-a-chip in the center, which is visibly connected to four (nearly) adjacent test pads. Looking closely, I noticed the four pads are labeled: RX, TX, VCC and GND. I recognized this as a standard UART (Universal Asynchronous Receiver-Transmitter) interface, which can be used to transfer data to and from the device via a console. Manufacturers often include these interfaces on embedded devices for development and debugging. While UART allows full-duplex serial communication using two data lines, it does not use a clocking line; therefore, both UART devices must agree on a communication speed, or baud rate.

To interface with the UART port from my computer I used a Shikra .[14] The Shikra is a USB device capable of communicating using several low-level protocols including JTAG, SPI and I2C. I soldered wires to all four UART pads (although VCC was left unconnected) as shown below, and connected RX, TX and GND to the Shikra according to the UART pinout provided by Xipiter.

Figure 13: UART Test Pads

[14] The Shikra. https://int3.cc/products/the-shikra.

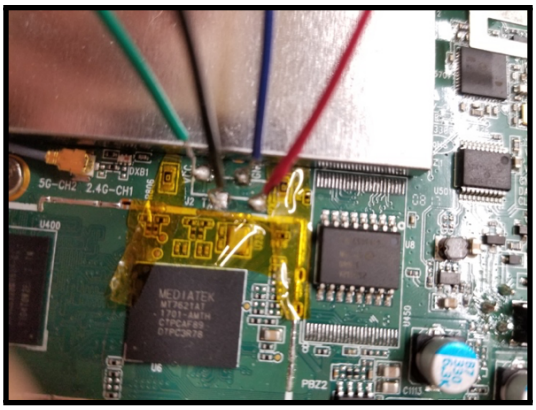

Figure 14: Soldered Leads

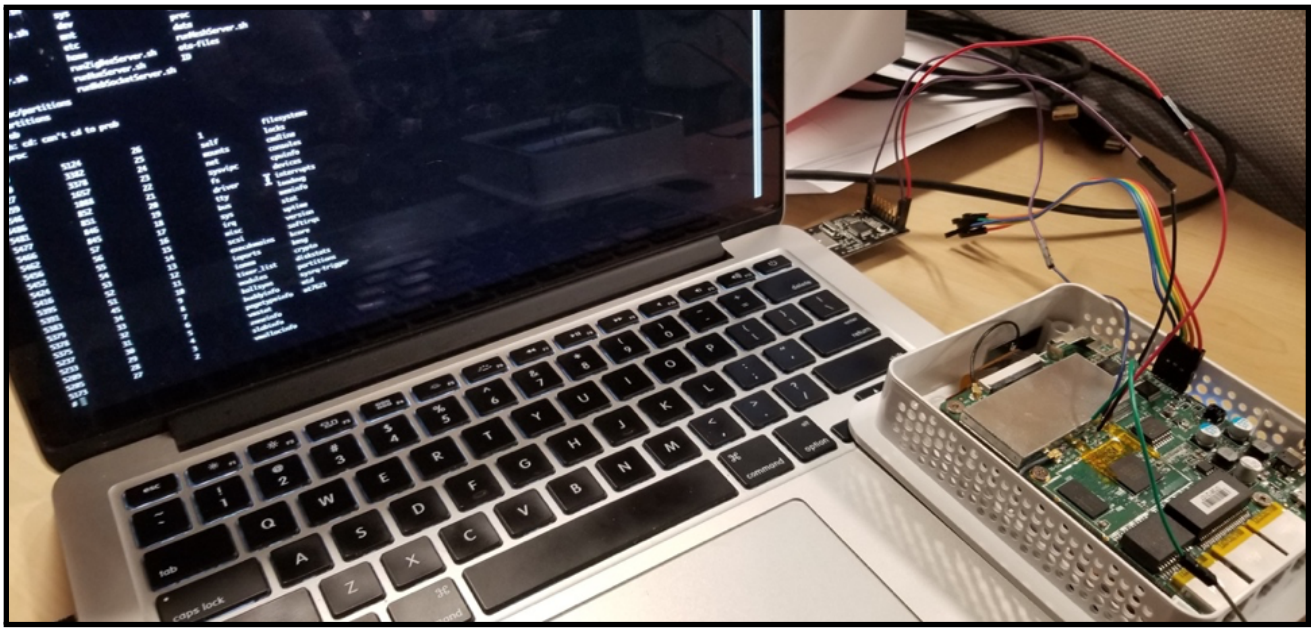

Figure 15: Using the Shikra

The next step was to find the baud rate. One method is to use a script that cycles through common baud rates and uses trial and error until human-readable output is observed. Another way is to manually probe the TX pad using an oscilloscope and observe the wavelength of the signal being transmitted; the baud rate (referred to as the 'frequency' in signal processing) can then be calculated as the inverse of the wavelength. Using the second method, I found that the baud rate is 57600 bps (bits per second).

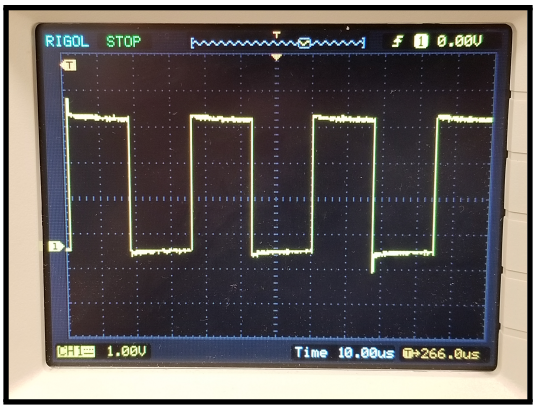

Figure 16: Finding the baud rate

Having calculated the baud rate, I connected to the interface to monitor the communication from the terminal using the 'screen' command as shown.

```
screen /dev/tty.usbserial-1420 57600
```

When I powered on the Almond I observed the boot process, and learned that the bootloader is Ralink UBoot[15] version '5.0.0.0'. Once the hardware was initialized I was then given several boot options as shown below.

```
Please choose the operation:
   1: Load system code to SDRAM via TFTP.
   2: Load system code then write to Flash via TFTP.
   3: Boot system code via Flash (default).
   4: Entr boot command line interface.
   7: Load Boot Loader code then write to Flash via Serial.
   9: Load Boot Loader code then write to Flash via TFTP.
```

Figure 17: Boot Menu

At this point my goal was to obtain shell access or find a way to extract the firmware. I did not want to load my own OS or bootloader, so only options 3 or 4 sounded promising. I decided to start with the boot command line interface, hoping it might allow me to dump the flash. I was given a UBoot prompt with the following options:

[15] u-boot - Github. https://github.com/u-boot/u-boot.

```
4: System Enter Boot Command Line Interface.

U-Boot 1.1.3 (Feb  8 2017 - 10:13:48)
MT7621 # ?
?       - alias for 'help'
bootm   - boot application image from memory
cp      - memory copy
erase   - erase SPI FLASH memory
go      - start application at address 'addr'
help    - print online help
loadb   - load binary file over serial line (kermit mode)
md      - memory display
mdio    - Ralink PHY register R/W command !!
mm      - memory modify (auto-incrementing)
nm      - memory modify (constant address)
printenv- print environment variables
reset   - Perform RESET of the CPU
rf      - read/write rf register
saveenv - save environment variables to persistent storage
setenv  - set environment variables
spi     - spi command
tftpboot- boot image via network using TFTP protocol
version - print monitor version
MT7621 #
```

Figure 18: UBoot CLI

The only command that seemed interesting here was 'spi', which is used to send commands to the flash chip via the SPI bus (see SPI section below for more details).

```
MT7621 # help spi
spi spi usage:
  spi id
  spi sr read
  spi sr write <value>
  spi read <addr> <len>
  spi erase <offs> <len>
  spi write <offs> <hex_str_value>

MT7621 # spi id
device id: 20 ba 20 7b 1c
MT7621 # spi read 00000000h 13
read len: 19
ff 0 0 10 0 0 0 0 fd 0 0 10 0 0 0 0 97 4 0
```

Figure 19: SPI Commands

Although I could read from (or write to) flash memory, there was no way for me to save the output on my computer as a firmware image. Before moving on, I checked the device/manufacturer ID and read out the first few bytes of nonvolatile memory (when interfacing directly with the flash chip later, I was able to use these outputs to confirm my results).

Having tried the boot CLI, I restarted the device and this time let the system boot from flash (as it does by default). I then observed the system booting an lzma-compressed MIPS Linux kernel, and was soon given a standard shell prompt. Having obtained a shell the first thing I did was check my user ID, which was '0' despite the account being named 'admin' instead of root, in addition to the 'etc/passwd' file.

```
# id
uid=0(admin) gid=0(admin)
```

Figure 20: Linux UID/GID

```
# cat /etc/passwd
admin:ezg5SWJ8ImIGk:0:0:Adminstrator:/:/bin/sh
```

Figure 21: Password File

Next, I explored the filesystem to find any sensitive information that might be useful to an attacker. Within the '/etc' directory I found the wi-fi password and the credentials for logging into the web application.

```
# cat /etc/Wireless/iNIC/iNIC_ap.dat | grep 'PSK'
AuthMode=WPA2PSK
WPAPSK=
WPAPSK1=wifi_pa$$word!
WPAPSK2=uz3QVWoS
WPAPSK3=Almond55328
```

Figure 22: Wi-Fi Password

```
# cat /etc/lighttpd.user
admin:admin
```

Figure 23: Web Dashboard Credentials

Within the same directory I found a file called 'services', which indicates that the devices is running Layer 2 Tunneling Protocol over port 1701. Checking with a quick Nmap scan I found that the port is indeed open, possibly exposing a remote DoS vulnerability.

```
# cat /etc/services
l2tp 1701/tcp l2f
l2tp 1701/udp l2f
```

Figure 24: L2TP running on port 1701

I also found a few things stored in the '/tmp' directory, including a configuration file containing the address of the DNS server; this could provide a potential attack vector, as pointing DNS to an attacker-controlled address could result in DNS cache poisoning (this could also be effectively done with the '/etc/hosts' file). Also in this directory are several database files; in one of these files, called 'apdb', I found the secret PIN required to unlock the touchscreen, as well as the administrator email address associated with the Almond's mobile application (which happens to be mine).

```
# cat /tmp/apdb
[securifi]
14@A@B
`AlmondMode2 NULL@Tf
Subscriptions1,0,0A@B1234)@@AlmondTamper0e@BrenTimeout30@z6@ArmedLocationsNone@@@%
`AlmondMode2 jrabinowitz2@gmail.com#
```

Figure 25: Unlock PIN & admin's email address

The '/tmp' directory also contains several log files. Looking through the cloud daemon log I found the Securifi server that the Almond connects to for checking firmware updates, diagnostic information, etc. I did not attempt to test this server, as it is out of scope; however, an attacker can use this information to impersonate the Securifi server and trick the Almond into a malicious connection. Another possibility would be to sniff the cloud traffic to capture the firmware that gets sent before/during an update, essentially extracting the firmware 'OTA' rather than from the hardware. I found a file in the '/almond' directory called 'cacert.pem', which contains a bundle of SSL certificates/keys; since the LAN traffic is all unencrypted these certificates are likely used for communicating with the Securifi cloud server.

```
# cat /tmp/CloudDaemon.log
[2019-8-22 7:47:33.11767]       {PRINT}         Almond+ Daemon started.
[2019-8-22 7:47:33.12705]       {INFO}  Current state : NOT_CONNECTED
[2019-8-22 7:47:33.12867]       {CRITICAL}      ** Using Default cloud server**
[2019-8-22 7:47:33.12881]       {INFO}  almond.securifi.com
[2019-8-22 7:47:33.23459]       {INFO}  3GBackup is not set
[2019-8-22 7:47:33.28989]       {INFO}  Internet Connection is there
```

Figure 26: Securifi Cloud Server

Within the '/bin' directory I found mostly standard BusyBox executables, in addition to a Samba server, ProFTPD (likely used for the web dashboard's FTP service), and the 'lighttpd' web server. To go further with the assessment I could start the Samba server and the FTP server (which can also be enabled from the web dashboard) and enumerate them using Nmap to check if they're associated with any known vulnerabilities. As I ran short on time due to the breadth of this assessment, I decided to leave this for further research. In '/usr/bin' I found 'tftp', a TFTP server, which I also left for future testing.

Having combed through most of the filesystem's contents, the one item that remained was reverse engineering the device's proprietary software to find any vulnerabilities that could be exploited remotely. I found a large handful of "interesting" binaries within the '/almond' directory, in addition to several proprietary shared object files within the '/lib' directory. To perform binary analysis I needed to obtain an image of the firmware, as the Almond does not have the

proper tools for reversing or debugging. Although it has 'dd', which can be used to create an image of the filesystem, it doesn't have netcat, telnet, ssh or any other tools I could use for remote extraction or copying.

SPI

As I was unable to obtain a firmware image through the UART interface I decided to try and extract it from the flash chip, as it is stored in nonvolatile memory and loaded into RAM during boot. The primary method of firmware extraction is chip-off, which entails desoldering with a heat gun, connecting the chip to a socket and connecting the socket to a chip reader or programmer to extract the contents. However, removing the chip can result in permanent damage, and moreover further testing/operation would require carefully reattaching the chip. Reviewing the flash chip's datasheet I noticed it has SPI capability. SPI (Serial Peripheral Interface) is a low-level synchronous serial protocol used for communication in embedded systems, and it operates in full duplex mode using a master-slave architecture. Through the SPI interface it is possible to read or write to the flash chip without removing it from the PCB.

To connect to the chip I used a 16-pin SOIC clip/socket with a breadboard, which I connected to my computer using a RedBoard[16] (a generic, Arduino Uno-compatible microcontroller by SparkFun Electronics). Rather than using the Shikra as I did with UART, I decided that developing my own hardware Swiss army knife would be a fruitful learning exercise. I connected the components according to the pinout and signal descriptions from the datasheet shown below. (Note: with this setup I did not need to power on the device to communicate with the chip, since my circuit provided power on its own. With the device on, there's a chance the processor will prevent SPI communication as the flash chip is already in use; this requires an analog workaround such as manually holding the processor in reset.)

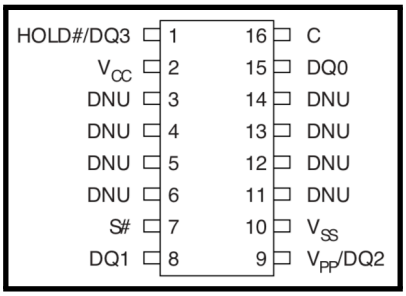

Figure 27: SPI Pinout

Flash Chip	RedBoard
C	CLK
S#	CS
DQ0	MOSI
DQ1	MISO
HOLD#	+3.3v
Vcc	+3.3v
Vpp	+3.3v
Vss	GND

[16] SparkFun RedBoard - Programmed with Arduino. https://www.sparkfun.com/products/13975.

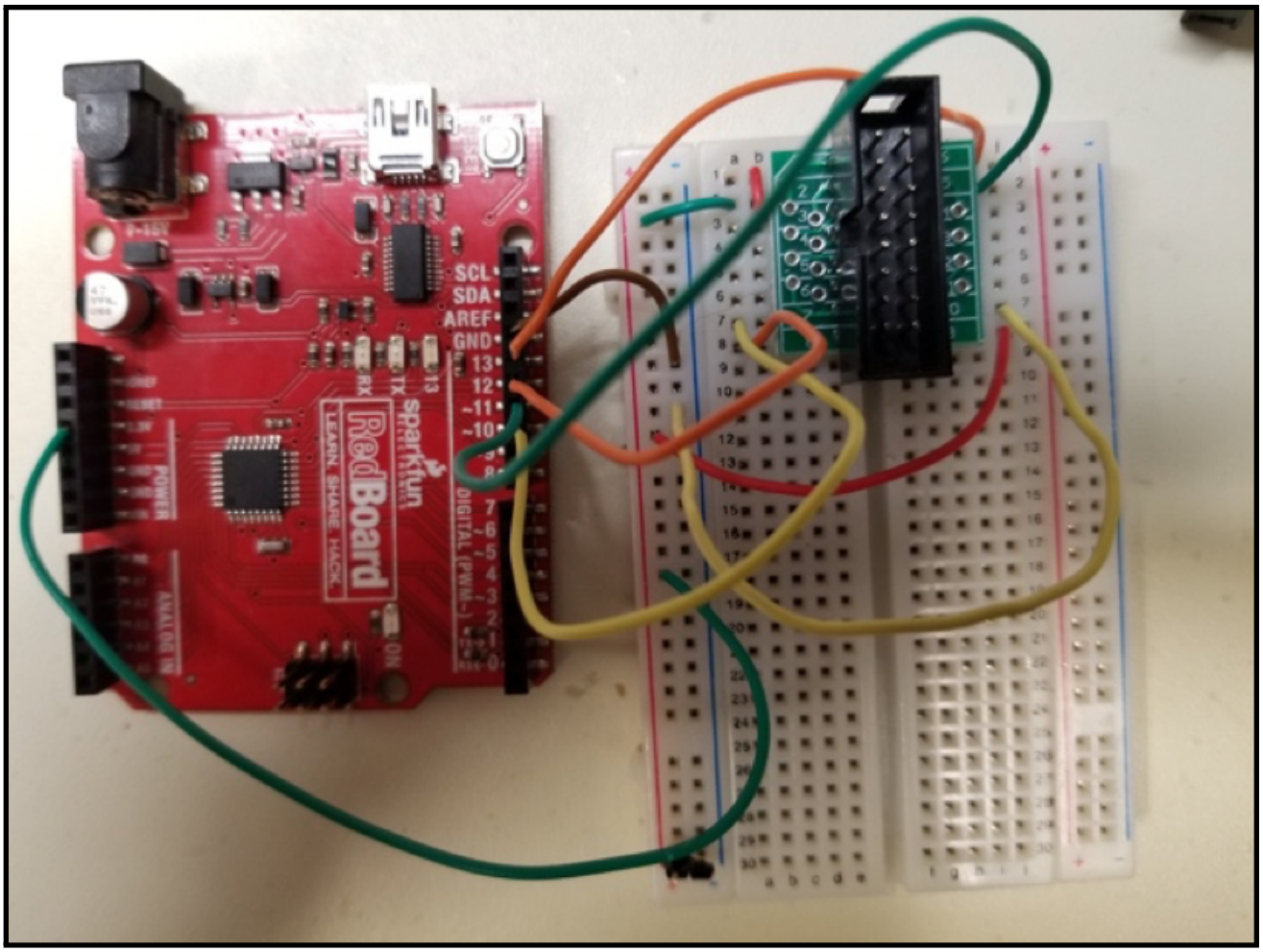

Figure 28: Pin Connections (Detailed View)

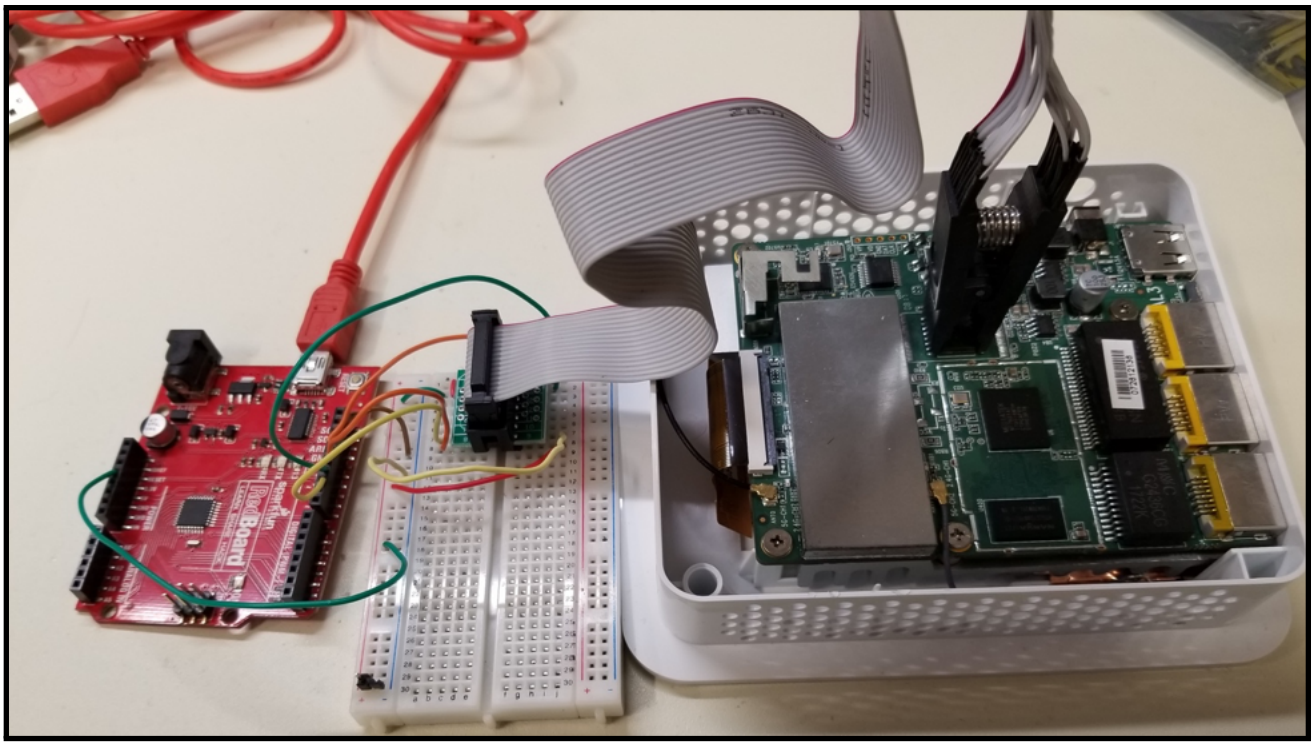

Figure 29: DIY Chip Reader

With my homemade chip reader set up I decided to test whether or not my device worked before attempting to extract the nonvolatile memory. A good way of testing this is to try and read out the device/manufacturer ID, as this is a quick operation and it proves the chip can receive and execute commands properly. To send SPI commands I first had to set a few parameters based on the datasheet including the clock rate, the SPI mode (which determines clock polarity/phase), and the bit shifting order. The code I wrote for reading the device/manufacturer ID can be found in the SOURCE CODE section. When executed it opens a serial connection to the microcontroller in the terminal, sends the 'RDID' command (as the appropriate hexadecimal byte), receives the response from the chip, and prints it to the screen. Once my code was verified and compiled I uploaded it to the RedBoard and executed it from the terminal using the command shown below.

```
screen /dev/tty.usbserial-DA011HJI 9600
```

I compared the output to the values found in the datasheet, confirming that my chip reader worked. The first non-empty byte is the Manufacturer ID, while the next two bytes make up the Device ID (comprised of the Memory Type and the Memory Capacity). In addition to the datasheet, I was able to match this with the output from the 'spi' command I used in the UBoot shell earlier.

```
Initializing...
Beginning SPI Transaction...
Sending data...
Received Bytes:
0
20
BA
20
```

Table 21: Read ID Data Out

Size (Bytes)	Name	Content Value	Assigned by
1	Manufacturer ID	20h	JEDEC
2	Device ID		
	Memory Type	BAh	Manufacturer
	Memory Capacity	20h (512Mb)	

Knowing my device was successful, I decided to check one more thing before moving on to the flash dump: the OTP array. OTP (one-time-programmable) memory is a special type of nonvolatile memory that can only be written to once, as programming after manufacturing works by "blowing" the fuses that correspond to bits and the process is irreversible. For this reason, embedded device manufacturers often use the OTP array for storing security information such as encryption keys, checksums, etc. They can also be protected from read operations using a control bit in the chip's status register. The block diagram below shows the 64-byte OTP array located outside of the main memory array.

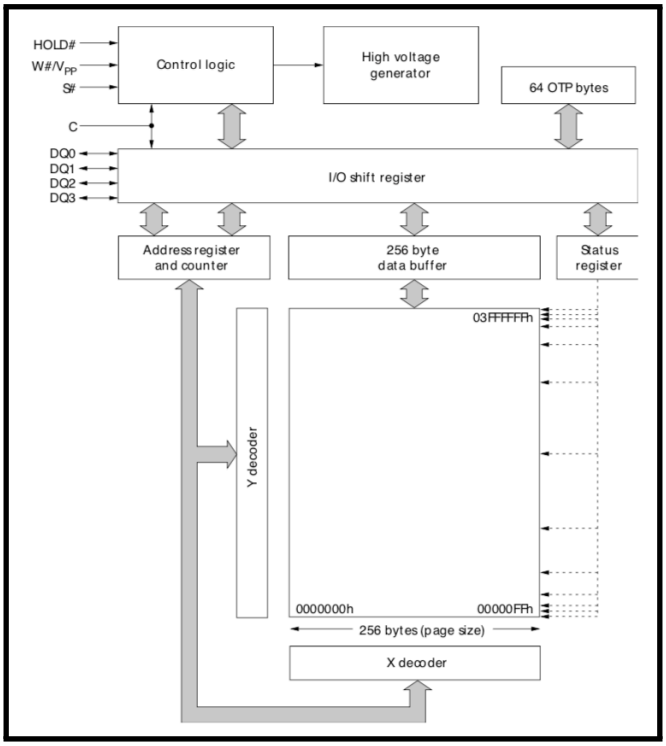

Figure 30: Flash Chip Block Diagram

To check if Securifi hid anything interesting in the OTP array I wrote another small piece of code, which can be found in the SOURCE CODE section. I ran this code using the same command, resulting in the output shown below. Typically, OTP memory comes with all bits initially set to "1"; in this case it appears Securifi did not use or modify the array, which is why the hexadecimal output appears as 64 "F"s (indicating an empty array).

```
Initializating...
Beginning SPI Transaction...
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Figure 31: Empty OTP Array

The next step was to actually dump the contents of the main memory array. Micron's command set offers several ways to do this. The most obvious way is to iterate through each address in memory and use the 'READ' command to obtain the data stored at each location. Alternatively, Micron offers a 'FAST READ' command that takes in a single address and automatically increments the address after reading from each location, allowing the entire array to be read out using a single command. In addition to standard SPI the chip is capable of 'DUAL'/'QUAD' output modes, which use additional data lines (for a total of two or four output lines, respectively) for faster read operations. I decided to stick

with the standard, however, since my circuit was already set up.

I wrote a program for my chip reader that sends the 'FAST READ' command along with the array's starting address (0x00000000) and then prints the data to the screen as it's received. The code can be found in the SOURCE CODE section. When the end of the array is reached, according to the datasheet, the operation wraps around and starts back at the first address. To prevent this unwanted behavior I used a counter in my code that stops the read operation once the array has been read from beginning to end. I also used the 'screen' command's logging feature this time, as shown below, in order to save the resulting output to a file.

```
screen -L /dev/tty.usbserial-DA011HJI 9600
```

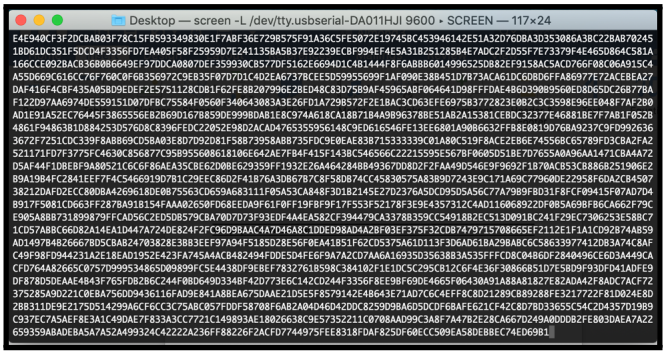

Figure 32: Output from 'FAST READ'

When I ran my code the flash memory contents were output to the terminal as expected, and they matched perfectly with the first few bytes I had checked earlier. However, the output was sent as hexadecimal byte representations of the data, rather than the data itself. So, the resulting log file I ended up with was only recognized by my computer as a giant string of human-readable ASCII characters. In order to obtain a valid image of the Almond's firmware I needed to convert these hexadecimal characters to raw data bytes. To that end I wrote the following quick Python script:

```
import binascii

with open('/Users/josh/Desktop/screenlog.0','r') as fp:
        text = fp.read()
        text = text.strip()
data = binascii.unhexlify(text)
with open('/Users/josh/Desktop/flashdump.bin','w+') as fp2:
        fp2.write(data)
```

Firmware Analysis

Having obtained the firmware from the flash chip, the next step was to perform firmware analysis in order to extract the filesystem for further digging. For a detailed look at the firmware analysis methodology I used see section FIRMWARE ANALYSIS . After a handful of steps I wound up with a copy of the Almond's filesystem, the structure of which is pictured here.

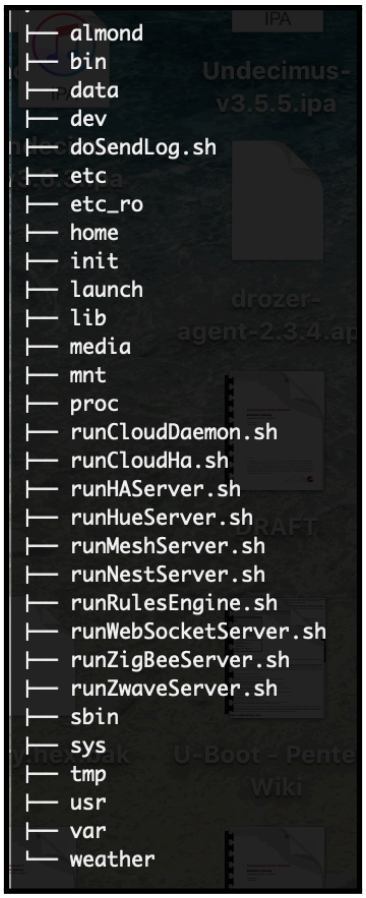

Figure 33: Almond 3 Filesystem

Binary Analysis

The final step in my analysis was to check some of Securifi's proprietary binary executables, which were primarily found within the "almond" directory of the device's filesystem. I reverse-engineered these binaries using Ghidra with the language set as "MIPS:LE:32". In order for Ghidra to recognize all symbols and imports I imported the device's filesystem as a whole, then analyzed the binaries individually. Due to time constraints, however, I was unable to find any vulnerabilities during this phase of the analysis. Based on flaws discovered in past iterations of the device there are likely several buffer overflow vulnerabilities, particularly contained within the web and network-related binaries (e.g. "webServer", "zigbee_server", "CloudDaemon", etc.). As such, this area would be the most ideal target for further research.

WEB

Authentication

The web dashboard authenticates users using Basic Authentication, and login requests are not encrypted. Therefore, anyone positioned on the network can capture the login credentials in Base64 and decode them to gain administrative control over the network.

Information Disclosure

The web application's source code contains a number of hardcoded secrets, one of which reveals the server is running an outdated version of Lighttpd. The server also utilizes SSI's (server-side includes) to load pages faster and CGI scripts for communicating in real-time with the Almond device and network peripherals; as these scripts are included in client-side HTML, the user can deduce backend functionality, allowing an attacker to query the API endpoints for sensitive information.

Clickjacking

The web application does not set HTTP headers 'X-Frame-Options' or 'Content-Security-Policy' to prevent framing attacks. Moreover, the application uses basic 'frame-busting' code which is trivial to bypass, leaving the dashboard user vulnerable to clickjacking/ UI-redressing attacks.

Cross-Site Request Forgery (CSRF)

The web application does not protect against cross site request forgery (CSRF). An attacker can generate a cross-site request in order to perform arbitrary actions on behalf of the user, such as adding or modifying network sensors/rules, or even log the user out of the application.

WebSocket

The dashboard communicates in cleartext with a WebSocket listening on port `8080`. An attacker positioned on the network can capture the WebSocket's secret key in order to send it commands and obtain sensitive device information or 'update' the Almond's mode of operation.

Firmware Upload

An attacker with access to the dashboard has the ability to flash the Almond's nonvolatile memory with modified firmware, containing some form of malware or back door. An attacker could also upload an invalid firmware image that passes the 'magic bytes' check, effectively bricking the device and resulting in a DoS (Denial of Service).

Remote Code Execution

The dashboard has a Console feature that allows sending arbitrary commands to the Almond's underlying OS. Commands are sent within 'command' parameter of a POST request to a CGI script called 'adm.cgi' in the 'cgi-bin' directory. Combined with the captured credentials, an attacker could use this capability to execute system commands remotely within the context of the web application. As the Almond OS comes with only one user account- 'admin' (root) –this effectively gives the attacker a root shell on the device.

MOBILE

Data Storage

Within the application's data directory are several database files storing sensitive information in plaintext, including web dashboard credentials, Wi-Fi SSIDs and passwords, WebView caches, credit card and autofill data, the PIN to unlock the Almond touchscreen, and emails/passwords for Almond user accounts. An attacker with physical access to a user's phone can use this information to take control of the user's account as well as the Almond device.

Source Code contains Hardcoded Secrets

The application's source code contains a large number of hardcoded secrets including API keys, links to private web resources/servers, session tokens, and the contents of a '.pem' file (SSL certificate) stored within a static string variable,

which is used for remote encrypted connections to Securifi's cloud servers.

Lack of Certificate Pinning

The application does not utilize SSL certificate pinning (or validation). As such, an attacker can conduct a man-in-the-middle attack and intercept the application's network traffic if they are able to install a malicious certificate on the user's device and obtain a privileged network position.

Remote Server Impersonation

The application sends queries to a remote server at 'https://utils.securifi.com' to obtain information on recent activity from the Almond network. These queries include an Oath Bearer token which an attacker can use to authenticate against the server. Combined with the lack of SSL certificate pinning, an attacker can capture these requests and the server's responses to obtain sensitive network and device information; moreover, the attacker can impersonate the remote server.

Application Supports Unmaintained Versions of Android

While the application targets Android SDK version 28, according to the manifest file, the minimum supported version is SDK version 16. Supporting versions of Android that no longer receive security updates weakens the overall security posture of the application. The oldest version of Android currently maintained by Google is 7.0 ("Nougat") which corresponds to SDK version 24.

HARDWARE

Unprotected UART

In order to prevent physical access to hardware components, manufacturers often utilize some form of tamper resistance such as steel enclosures, locks, epoxy, etc. As Securifi did not implement any such protection the board's UART is exposed through the four test pads, allowing an attacker with physical access to interface with the device via a privileged command shell and interact directly with the OS. Securifi could also have protected UART access by disabling through software (i.e. with a boot delay).

Microcontroller lacks Read/Copy Protection

To prevent an attacker from extracting the contents of the PIC microcontroller and in turn copying or reverse engineering the software, firmware developers can turn on PIC's code protection bit, which does not allow chip reads when asserted. Another method, Unique ID Authentication, requires that the application verifies itself against a unique ID before launching. Finally, removal of the chip's numbers could prevent an attacker from obtaining the pinout required to interface using ICSP. Securifi did not implement any such protection for the Microcontroller onboard the Almond device, leaving it susceptible to tampering or reverse-engineering and exploitation.

While this assessment yielded multiple vulnerabilities, it is likely that more flaws remain uncovered. With such a large and varied attack surface it would take a substantial amount of time to exhaust all potential attack vectors. I came across several items during my research that necessitate further exploration. Listed below are some areas that could serve as jumping off points for continued research:

Fuzzing the WebSocket

Develop a custom fuzzer to test the WebSocket listening on port 7681. Rules/grammar can be based on the client commands in the intercepted WebSocket messages. I found a tool called 'socat'[17] that can be used as a WebSocket client to craft messages and send them directly to the server as queries:

```
admins-MacBook-Pro:~ user$ socat - TCP:192.168.7.183:7681,crnl
GET /admin:admin HTTP/1.1
Host: 192.168.7.183:7681
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://192.168.7.183
Sec-WebSocket-Key: IsRPTOiJ88Ae2zUrQd9s5w==
Connection: keep-alive, Upgrade
Cookie: selectedTab=AC3tabs-3
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket

HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: /kibRBqROlfgQX3VonHtfSLV7nU=

?F{"CommandType":"DynamicAlmondModeUpdated","Mode":"2","EmailId":"NULL"}
```

Figure 34: WebSocket Interface

Many similar open-source tools exist to perform black or gray box tests against WebSocket servers.

Server Vulnerabilities

Test the web application against the list of associated lighttpd vulnerabilities found on Security Focus.[18] The version of lighttpd running beneath the dashboard application is 1.4.20, which is quite out of date. I did not check which of these CVE's have ready-made proof-of-concept exploits already available.

[17] craSH. socat - Github. https://github.com/craSH/socat.
[18] SecurityFocus. https://www.securityfocus.com/bid.

Figure 35: Lighttpd version 1.4.20 vulnerabilities

File Upload Attacks

Assess whether or not the dashboard's file upload form could be tricked into updating the firmware with an invalid image file (or a completely different file type). In addition to bricking the device with a DoS it may be possible for an attacker to upload a custom/modified Almond firmware image, containing either a backdoor or some sort of malicious software.

"Busting" Frame-Busting

Several of the pages on the web dashboard are frameable, and the application does not utilize X-Frame-Options or Content-Security-Policy HTTP headers as appropriate. Find a bypass for the frame-busting code used on the client side to prevent clickjacking/framing attacks- preferrably one that is browser independent. Possible bypasses include:

- Double framing
- Disabling JavaSript
- onBeforeUnload event

Enabled FTP/TFTP Servers

The web dashboard offers FTP server functionality which can be remotely enabled or disabled. The FTP server should be enumerated and/or fuzz-tested, as it may provide an entry point for an attacker positioned on the local network. Note: this can also be enabled from the shell using 'proftpd'. Also found from the shell was a 'tftp' server.

Samba Server

One of the executables stored on the device is a script for configuring and running a Samba server over port 445. As the web server is out of date, there is a good chance this server is as well; this would be a good network based attack vector to look into.

Impersonating Securifi

I found that the Almond connects to a cloud server at almond.securifi.com. Although this server is outside of the scope for this assessment, an attacker could impersonate the Securifi server to capture any traffic the Almond sends out to the cloud. Another possible route would be to capture the OTA firmware update sent by this server, thereby obtaining

the firmware remotely.

iOS Application

For the sake of time I only tested the Android version of the mobile application. In the future it would be worth testing the iPhone application as well, which can be downloaded from the Apple store.

Conclusion

For decades many have dreamed of one day living in a fully automated home. At one point this was merely something seen in science fiction; now, products like the Almond 3 are turning this dream into a reality. But, with the Internet of Things still in its infancy, most companies producing IoT-enabled smart devices are in a rush to release their products before their competitors can corner the market. They also tend to try and equip these products with as much functionality as possible, leaving plenty of room for bugs and misconfigurations while security is left as an afterthought.

Through some thorough testing, I found that Securifi's Almond 3 is no exception to this pattern. Overall, my assessment uncovered a substantial handful of security vulnerabilities contained within the Almond 3 and its various elements. These vulnerabilities, when considered collectively, put the customer's home network at great risk of being compromised by an attacker. The risk is compounded by the fact that so many everyday objects and household items are starting to be replaced by IoT devices.

In hindsight, there were many security issues and conventions the Securifi team made sure to consider. However, products such as this are relatively complex in nature, and the broad set of features leads to a large attack surface that is nearly impossible to defend. In assessing the web/mobile applications, the low energy network communications, and the hardware of the device itself, I learned a great deal about security engineering/testing, embedded devices, and IoT security. Further research into these components would provide an opportunity to learn even more.

For a future project, it would be useful to take the contents of this report and compile a document that details a formal methodology for testing embedded devices and IoT systems. Alternatively, this report could be used to craft a best practices document for helping IoT manufacturers protect their designs by incorporating security within the product development lifecycle.

References

[api] riverloopsec. apimote –Github. https://github.com/riverloopsec/apimote.

[apk] ibotpeaches. Apktool - Github. https://ibotpeaches.github.io/Apktool/.

[app] Apple App Store. Almond by Securifi. https://apps.apple.com/us/app/almond-by-securifi/id908025757.

[bin] Visual analysis of binary files. http://binvis.io.

[bur] Portswigger- Burp Suite. https://portswigger.net/burp.

[cat] craSH. socat - Github. https://github.com/craSH/socat.

[d2j] pxb1988. dex2jar –Github. https://github.com/pxb1988/dex2jar.

[drz] MWR Labs - Drozer. https://labs.mwrinfosecurity.com/tools/drozer/.

[foc] SecurityFocus. https://www.securityfocus.com/bid.

[gps] Google Play Store. Almond. https://play.google.com/store/apps/details?id=com.securifi.almondplus&hl=en_US.

[gui] jd-gui –Github. https://github.com/java-decompiler/jd-gui.

[int3] The Shikra. https://int3.cc/products/the-shikra.

[kil] riverloopsec. killerbee –Github. https://github.com/riverloopsec/killerbee.

[Mic11] Micron. N25Q512A Datasheet. https://www.micron.com/products/nor-flash/serial-nor-flash.

[red] SparkFun RedBoard - Programmed with Arduino. https://www.sparkfun.com/products/13975.

[sec] *How to Update your Almond?* https://www.securifi.com/software-update.

[stripe] Stripe. https://stripe.com.

[Tek13] MediaTek. MT7621 Datasheet. http://www.t-firefly.com/download/FireWRT/hardware/MT7621.pdf.

[ubt] u-boot - Github. https://github.com/u-boot/u-boot.

SOURCE CODE

The following code was used to read out the device/manufacturer ID from the SPI flash chip. It does so by sending the 'RDID' command bytes while the CS pin is driven low, receiving the response and then printing them to the serial monitor.

```
#include <SPI.h>
#define RDID 0x9F
#define CLOCKRATE 85000000

const int csPin = 10;
const int mosiPin = 11;
const int misoPin = 12;
const int clkPin = 13;

byte transferAndWait (const byte what)
  {
  byte a = SPI.transfer (what);
  delayMicroseconds (20);
  return a;
  }

void setup(){
  Serial.begin(9600);
  Serial.println("Initializating...");
  SPI.begin();
  pinMode(csPin, OUTPUT);
  pinMode(mosiPin, OUTPUT);
  pinMode(misoPin, INPUT);
  pinMode(clkPin, OUTPUT);
  delay(100);

  Serial.println("Beginning SPI Transaction...");
  SPI.beginTransaction(SPISettings(CLOCKRATE, MSBFIRST, SPI_MODE0));
  byte a,b,c,d;

  //send data (output '9F')
  digitalWrite(csPin,LOW);
  Serial.println("Sending data...");

  a = transferAndWait(RDID);
  b = transferAndWait(0x00);
  c = transferAndWait(0x00);
  d = transferAndWait(0x00);

  digitalWrite(csPin,HIGH);
  SPI.endTransaction();
  SPI.end();

  Serial.println("Received Bytes:");
  //Serial.println(a, HEX);
  Serial.println(b, HEX);
  Serial.println(c, HEX);
  Serial.println(d, HEX);
```

```
}

void loop(){
}
```

The following code was used to read out the contents of the OTP (one-time-programmable) array in the SPI flash chip. It does so by sending the 'RDOTP' command bytes while the CS pin is driven low and then iterating through the 64 bytes of the array, printing each one to the serial monitor.

```
#include <SPI.h>
#define RDOTP 0x4B
#define CLOCKRATE 85000000

const int csPin = 10;
const int mosiPin = 11;
const int misoPin = 12;
const int clkPin = 13;

byte transferAndWait (const byte what)
  {
  byte a = SPI.transfer (what);
  delayMicroseconds (20);
  return a;
  }

void setup() {
  Serial.begin(9600);
  Serial.println("Initializating...");
  SPI.begin();
  pinMode(csPin, OUTPUT);
  pinMode(mosiPin, OUTPUT);
  pinMode(misoPin, INPUT);
  pinMode(clkPin, OUTPUT);
  delay(100);

  Serial.println("Beginning SPI Transaction...");
  SPI.beginTransaction(SPISettings(CLOCKRATE, MSBFIRST, SPI_MODE0));
  byte data;
  int counter;
  digitalWrite(csPin,LOW);

  transferAndWait(RDOTP);
  transferAndWait(0x00);
  transferAndWait(0x00);
  transferAndWait(0x00);

  for (counter=0; counter<64; counter++){
    data = transferAndWait(0x00);
    Serial.print(data, HEX);
  }

  digitalWrite(csPin,HIGH);
```

```
  SPI.endTransaction();
  SPI.end()
}

void loop() {
}
```

The following code was used to read out the contents of the main memory array in the SPI flash chip. It does so by sending the 'FASTREAD' command bytes while the CS pin is driven low and then iterating through each address in memory, printing the value at each location to the serial monitor.

```
#include <SPI.h>
#define FASTREAD 0x0B
#define CLOCKRATE 108000000
#define BYTECOUNT 67108864
const int csPin = 10;
const int mosiPin = 11;
const int misoPin = 12;
const int clkPin = 13;

byte transferAndWait (const byte what)
  {
  byte a = SPI.transfer (what);
  delayMicroseconds (20);
  return a;
  }

void setup(){
  Serial.begin(9600);
  SPI.begin();
  pinMode(csPin, OUTPUT);
  pinMode(mosiPin, OUTPUT);
  pinMode(misoPin, INPUT);
  pinMode(clkPin, OUTPUT);
  delay(100);

  SPI.beginTransaction(SPISettings(CLOCKRATE, MSBFIRST, SPI_MODE0));
  byte data;
  unsigned long int counter;

  digitalWrite(csPin,LOW);

  transferAndWait(FASTREAD);
  transferAndWait(0x00);
  transferAndWait(0x00);
  transferAndWait(0x00);
  transferAndWait(0x00);

  for(counter = 0; counter < BYTECOUNT; counter++){
    data = transferAndWait(0x00);
    Serial.print(data, HEX);
  }
```

```
  digitalWrite(csPin,HIGH);
  SPI.endTransaction();
  SPI.end();
}

void loop(){
}
```

FIRMWARE ANALYSIS

Having converted the flash dump from hexadecimal to raw data, I ran the 'file' command to confirm I had a valid firmware image.

```
admins-MacBook-Pro:Almond3 user$ file AL3-R024-64MB
AL3-R024-64MB: u-boot legacy uImage, Linux Kernel Image, Linux/MIPS, OS Kernel Image (lzma), 1533035
➜  4 bytes, Mon Feb 12 08:30:40 2018, Load Address: 0x80001000, Entry Point: 0x80511C60, Header CRC:
➜  0xFFBED8BC, Data CRC: 0x34A82D5D
```

I also ran 'hexdump' and 'strings' to confirm; I recognized the first few bytes, *'27 05 19 56'*, as the file signature for a Das U-Boot Universal Boot Loader.

```
admins-MacBook-Pro:Almond3 user$ hexdump -C AL3-R024-64MB | head -n 20
00000000  27 05 19 56 ff be d8 bc  5a 81 50 b0 00 e9 ec 32  |'..V....Z.P....2|
00000010  80 00 10 00 80 51 1c 60  34 a8 2d 5d 05 05 02 03  |.....Q.`4.-]....|
00000020  4c 69 6e 75 78 20 4b 65  72 6e 65 6c 20 49 6d 61  |Linux Kernel Ima|
00000030  67 65 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |ge..............|
00000040  5d 00 00 00 02 80 a9 32  01 00 00 00 00 00 00 6f  |]......2.......o|
00000050  fd ff ff a3 b7 7f 96 03  24 66 35 e0 52 8b b5 ab  |........$f5.R...|
00000060  c8 df 26 60 5f c3 27 7f  06 d6 18 c9 28 08 9c 27  |..&`_.'.....(..'|
00000070  82 2f 66 59 d7 f3 62 e3  c8 8e 09 9f 3c 46 bd c3  |./fY..b.....<F..|
00000080  31 94 c5 ad 3e 6f 31 2d  a2 a2 14 e9 d5 b3 7e cc  |1...>o1-......~.|
00000090  cd fd 4f ac 52 83 9d 30  1c e6 26 ff 72 44 cc cf  |..O.R..0..&.rD..|
000000a0  a0 1a c6 8a be 68 ca 5f  52 5f 18 64 db 39 e5 18  |.....h._R_.d.9..|
000000b0  a5 dc e9 77 ae dc f4 b7  26 79 16 cc 76 f6 87 a2  |...w....&y..v...|
000000c0  e7 25 70 3c 8f 7a 77 a3  ee bd a7 f0 dc a4 ad a7  |.%p<.zw.........|
000000d0  46 38 b1 82 78 7f e6 dc  65 70 7f 92 65 60 78 75  |F8..x...ep..e`xu|
000000e0  bc b0 34 7b e6 08 b3 d3  20 67 bc 94 43 76 be 48  |..4{.... g..Cv.H|
000000f0  4b ee 3c 3c ee fc fb e6  0d 1b 12 eb 07 5d 86 fa  |K.<<.........]..|
00000100  09 67 be f5 4e 41 67 e6  28 7e ab 0e c0 76 24 0a  |.g..NAg.(~...v$.|
00000110  7c c0 0c aa 07 71 cc a5  dc a6 bf 2e 25 d4 17 4f  ||....q......%..O|
00000120  4f ef de ee 13 ef 9d 42  36 e4 2e 0e 37 6a 76 9e  |O......B6...7jv.|
00000130  40 83 1b 7a 13 51 ec 27  c2 c5 10 77 42 45 1d c4  |@..z.Q.'...wBE..|
```

```
admins-MacBook-Pro:Almond3 user$ strings -n 15 AL3-R024-64MB
Linux Kernel Image
8x[{!OKR/N6dy@k
(( S<!D\Wo/}12BL&
(...)
```

Next, I plotted the file's entropy using 'binwalk' as well as a web-based tool called binvis.io[19] to confirm the use of LZMA compression. Embedded devices such as the Almond 3 typically use compression to minimize nonvolatile memory

[19] Visual analysis of binary files. http://binvis.io.

requirements.

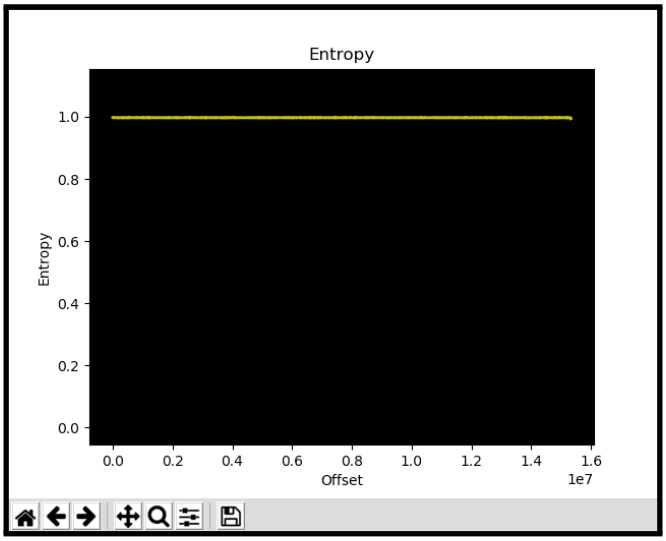

Figure 36: Entropy Plot

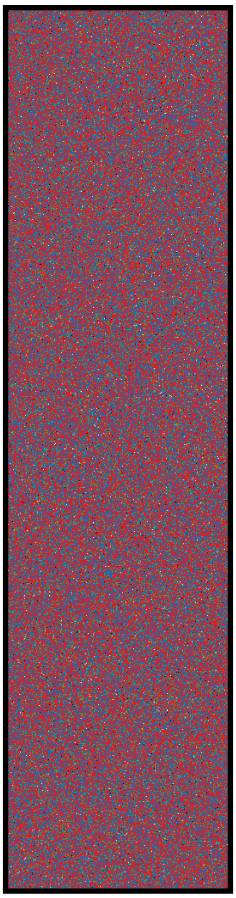

Figure 37: Visual Analysis

To finish my static analysis I used 'binwalk' once more to scan the image for the partitions and their offsets.

```
admins-MacBook-Pro:Almond3 user$ binwalk AL3-R024-64MB

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 6 4 bytes, header CRC: 0xFFBED8BC, created: 2
➜  018-02-12 08:30:40, image size: 15330354 bytes, Data Address: 0x80001000, Entry Point: 0x80511C60,
➜  data CRC: 0x34A82D5D, OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma,
➜  image name: "Linux Kernel Image"
64            0x40            LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes,
➜  uncompressed size: 20097408 bytes
2941210       0x2CE11A        MySQL ISAM index file Version 4
```

At this point I was only interested in the kernel, which contains the filesystem. With the firmware confidently mapped out I used the information from the previous command to carve out the kernel and then decompress it using 'lzma'. I then used 'binwalk' one last time to extract any known file types

```
admins-MacBook-Pro:Almond3 user$ dd if=AL3-R024-64MB of=kernel.lzma bs=1 skip=64
15330354+0 records in
15330354+0 records out
```

```
15330354 bytes transferred in 62.390698 secs (245715 bytes/sec)
admins-MacBook-Pro:Almond3 user$ lzma -d kernel.lzma
admins-MacBook-Pro:Almond3 user$ binwalk -e kernel

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
5361768       0x51D068        Linux kernel version "3.10.14 (bamboo@ip-10-69-60-34) (gcc version 4.6.3
➜  (Buildroot 2012.11.1) ) #960 SMP Mon Feb 12 13:59:56 IST 2018"
5361940       0x51D114        CRC32 polynomial table, little endian
5386312       0x523048        gzip compressed data, maximum compression, from Unix, NULL date (1970-01-
➜  01 00:00:00)
5724420       0x575904        SHA256 hash constants, little endian
5738575       0x57904F        Neighborly text, "NeighborRequestHandle_ActionHandle"
6481824       0x62E7A0        xz compressed data
6499640       0x632D38        Unix path: /lib/firmware/updates/3.10.14
...
(unix path strings, truncated for brevity)
...
6904192       0x695980        CRC32 polynomial table, little endian
6947676       0x6A035C        YAFFS filesystem
7069600       0x6BDFA0        CRC32 polynomial table, little endian
7142688       0x6CFD20        Unix path: /RAM/driver/fce/fce.c
7142796       0x6CFD8C        Unix path: /RAM/peri/mcuctl/mcuctl.c
7147404       0x6D0F8C        SHA256 hash constants, little endian
7185863       0x6DA5C7        Unix path: /core/lc/dl/include/dl_dev_impl.h
7496396        0x7262CC        LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes,
➜  uncompressed size: -1 bytes
```

One of the extracted files was a 'cpio' archive containing the Almond 3 root filesystem. From here I could explore the device's contents further and reverse engineer any proprietary binaries to find vulnerabilities that can be exploited remotely. The filesystem's overall structure is shown below.

```
admins-MacBook-Pro:Almond3 user$ cd _kernel.extracted/
admins-MacBook-Pro:_kernel.extracted user$ file ./*
./523048:    Linux make config build file, ASCII text
./62E7A0.xz: XZ compressed data
./7262CC:    ASCII cpio archive (SVR4 with no CRC)
./7262CC.7z: LZMA compressed data, streamed
admins-MacBook-Pro:_kernel.extracted user$ mkdir /tmp/cpio-root
admins-MacBook-Pro:_kernel.extracted user$ cp 7262CC /tmp/cpio-root/
admins-MacBook-Pro:_kernel.extracted user$ cd /tmp/cpio-root/
admins-MacBook-Pro:cpio-root user$ cpio -i < 7262CC
103619 blocks
```

```
admins-MacBook-Pro:cpio-root user$ ls -l
total 103720
-rw-r--r--    1 user  wheel  53052928 Aug  7 17:22 7262CC
drwxr-xr-x  360 user  wheel     11520 Aug  7 17:22 almond
drwxr-xr-x   99 user  wheel      3168 Aug  7 17:22 bin
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 data
drwxr-xr-x    3 user  wheel        96 Aug  7 17:22 dev
-rwxr-xr-x    1 user  wheel       676 Aug  7 17:22 doSendLog.sh
drwxr-xr-x    3 user  wheel        96 Aug  7 17:22 etc
```

```
drwxr-xr-x   19 user  wheel       608 Aug  7 17:22 etc_ro
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 home
lrwxrwxrwx    1 user  wheel        11 Aug  7 17:22 init -> bin/busybox
-rwxr-xr-x    1 user  wheel      2936 Aug  7 17:22 launch
drwxr-xr-x  107 user  wheel      3424 Aug  7 17:22 lib
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 media
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 mnt
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 proc
-rwxr-xr-x    1 user  wheel       853 Aug  7 17:22 runCloudDaemon.sh
-rwxr-xr-x    1 user  wheel       667 Aug  7 17:22 runCloudHa.sh
-rwxr-xr-x    1 user  wheel       522 Aug  7 17:22 runHAServer.sh
-rwxr-xr-x    1 user  wheel       535 Aug  7 17:22 runHueServer.sh
-rwxr-xr-x    1 user  wheel       814 Aug  7 17:22 runMeshServer.sh
-rwxr-xr-x    1 user  wheel       612 Aug  7 17:22 runNestServer.sh
-rwxr-xr-x    1 user  wheel       553 Aug  7 17:22 runRulesEngine.sh
-rwxr-xr-x    1 user  wheel       545 Aug  7 17:22 runWebSocketServer.sh
-rwxr-xr-x    1 user  wheel       576 Aug  7 17:22 runZigBeeServer.sh
-rwxr-xr-x    1 user  wheel      1328 Aug  7 17:22 runZwaveServer.sh
drwxr-xr-x   83 user  wheel      2656 Aug  7 17:22 sbin
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 sys
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 tmp
drwxr-xr-x    5 user  wheel       160 Aug  7 17:22 usr
drwxr-xr-x    2 user  wheel        64 Aug  7 17:22 var
drwxr-xr-x    3 user  wheel        96 Aug  7 17:22 weather
```