Android Apps Security

Mitigate Hacking Attacks and Security Breaches

—

Second Edition

—

Sheran Gunasekera

Apress®

Android Apps Security

Mitigate Hacking Attacks and Security Breaches

Second Edition

Sheran Gunasekera

Apress®

Android Apps Security: Mitigate Hacking Attacks and Security Breaches

Sheran Gunasekera
Singapore, Singapore

ISBN-13 (pbk): 978-1-4842-1681-1 ISBN-13 (electronic): 978-1-4842-1682-8
https://doi.org/10.1007/978-1-4842-1682-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. 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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

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/9781484216811. For more detailed information, please visit http://www.apress.com/source-code.

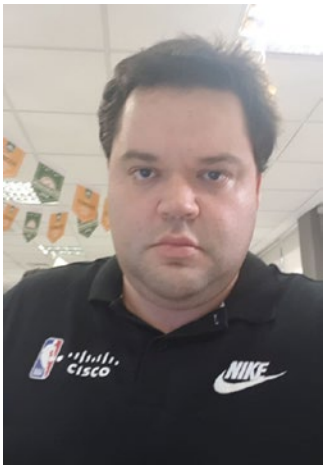
Printed on acid-free paper

Table of Contents

About the Author

Sheran Gunasekera is a security researcher and software developer. He is cofounder and Director of Research for Madison Technologies, a product development company in Singapore, where he advises the in-house engineering team in both personal computer and mobile device security. He is also one of the co-founders of RedStorm, an Information Security Bug Bounty Platform. Sheran's foray into mobile security began in 2009 when he started with BlackBerry security research. Since then, he has been in leadership roles in both engineering and security at several startups in Asia. He publishes research that he has done on his blog at `https://sheran.blog`.

About the Technical Reviewer



Thiago Magalhaes is a professional with more than a decade of experience in the information technology area with a wide experience in designing and dealing with large-scale distributed production environments. He is also a skilled DevOps engineer supporting, automating, and optimizing mission-critical deployments. A Linux lover by nature, he has a broad area of responsibility focused on security, high availability, reliability, and troubleshooting. He has hands-on experience in the administration and maintenance of application services like DNS, OpenLDAP, Samba, Mail, HTTP, Apache Tomcat, Squid, DHCP, SMTP, FTP, IMAP, NIS, and NFS. Last but not least, he is responsible for the ongoing maintenance, growth, and development of large-scale servers running Unix. In his spare time, he loves cooking for his friends and watching Netflix.

Acknowledgments

I would like to thank my family for their support and understanding during the authoring process. To my wife Tess, thanks for putting up with my absence and for waiting on me hand and foot while I wrote this book. To my daughter Shoshana, thank you for those times of laughter that I desperately needed and for your adultlike understanding of why I was doing what I was doing. My two furry, canine babies Zeus and Morpheus were instant stress reducers whenever they would somehow manage to get past Tess' guard and nudge my legs. They aren't at a high enough reading level to appreciate this note, but I'm sure they will get the message somehow.

I would also like to thank my friend and cofounder Prabu for essentially running our company entirely by himself as I dropped off the face of the earth to devote time to this book. Prabs, I don't tell you enough how much I appreciate what you have done for the company. Thank you for shouldering not only your but my responsibilities in running the company while I wrote this book.

To my friends and cofounders at RedStorm, Ariesto and Ele, thank you for keeping the InfoSec ship running smoothly while I was MIA.

To Thiago, my Technical Editor, thank you! I enjoyed your notes and insights as you reviewed the chapters in this book. It was great fun working with you.

Last but not least, team Apress. Thank you all for your help in making this second edition a reality. Mark Powers, I really enjoyed working with you. You know how to get the best out of an author, and I want to thank you for igniting the spark for me to keep researching even after this book. Steve Anglin, thank you for your patience and persistence. It took almost a decade, but it's finally done. The rest of the folks behind the scenes that I may have not met or spoken with, thank you!

Introduction

This book was a long time coming, and yet I can only feel that now was the perfect time to write it and publish it. Much like a young stand-up comedian who when just starting out has to collect all his life experience with which to deliver as humor, this second edition is a collection of personal experiences and research done along the way. This book is intended as a reference tool rather than an in-depth, granular teaching tool. It is a better friend to those developers and security researchers who are in their early-mid career than those just starting out. It is a collection of how I have done things and the reasons why I chose to do them the way I did.

In this book, I approach Android security from an offensive standpoint. If the first edition were the Blue Team, then this one is definitely the Red Team book. The principle I try to stand by in this book is that the best way to test your app is by breaking it and breaking it into as many pieces as you can. A true test of your app will be if it can withstand some of the techniques that we use in this book because it is a collection of techniques that are being used out there today. To this end, you will find a lot of information about how to intercept network traffic, how to break SSL and SSL Pinning, how to root your device, and then how to figure out that security is a lot more than looking for that silver-bullet piece of tech. It is never the case. You have to do the work. You have to research; you have to test and you have to understand the behavior – of apps and people. There is no silver bullet to security; you have to spend countless hours and, yes, sleepless nights worrying about it.

This book is also a work in progress I feel. As I wrote the chapters, I felt myself taken in different areas that I could not afford to explore. I hope to revisit some of those topics in the future and who knows? Maybe there will be another book. I do hope you find the book useful and that you learn to look at security from a different perspective. If there's one thing I want you to take away from this book, it is that you can't have security on autopilot. It is a topic you have to think about and consciously make decisions about at every step of the way. The bad guys out there will not rest, so that means less time to celebrate your wins and more time to spend looking at worst-case scenarios in your very own bubble of paranoia.

CHAPTER 1

Introduction

A little over seven years ago, I wrote the first edition to this book, and to be honest, I could have done better. So, when Apress reached out to me to refresh this book, I was elated – elated because not only do I get the opportunity to do better but also because in the space between the first and the second edition, I had the opportunity of taking a fantastic journey. It's a journey that had many highs and lows – one part where I led a team of software builders and one where I led a team of software breakers. I will give you an insight into how it feels to build and secure a product in a company which went from 40 people to over 3500 people seemingly overnight – a company that, as of writing this book, has a valuation of ten billion US $.

Before I take you on this journey, I would like to thank Steve Anglin at Apress who gave me that second chance at doing better and who still believed in me enough to reach out to. Together with me on this book is Mark Powers who I have worked with closely in making sure all parts of this book are good to go. Also, I'd like to thank all the folks on the team who I will not interact with directly, but who I know are there making sure this book looks its best. A big thank you to all of you.

The Startup Landscape

Most folks who have not been living under a rock are keenly aware of how the "startup ecosystem" has witnessed meteoric growth. I attribute this growth to the VCs and investors that have been pumping in hundreds of millions of dollars into round after round of fundraising. With more capital being available, it seemed like more startups popped up out of thin air. At the center of each startup was almost always an app. Be it an iOS or Android app, it existed and was mostly touted as a solution to a problem at your fingertips. Now this was not happening only in the United States, although it really took root there. Word of startups that built apps spread around the globe. Headlines proclaiming what seemed like colossal sums of money being raised by these startups

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_1

made everyone everywhere sit up and take note. Asia was no exception. It saw the success that the startups in the United States had and raced to keep pace. The startups were different, though. Asia (excluding China – because China's startup ecosystem is considerably different) had a different set of challenges, and thus the startups that were born there had a unique set of operating conditions that made it unique to Asia alone. Some challenges were a lack of stable infrastructure, poor handset quality and thus handset performance, and a more reserved culture when it came to trust, payments, and app usage.

Through it all, however, one key datapoint has remained a crucial success factor for an app waving startup: the user base, or the number of users of that product or app. For startups to attract large funding rounds, it had to show growth. One key metric that defined growth was weirdly not revenue. Instead, it was the number of users that interacted with the app. The more users you had, the more valuable you could become in the eyes of VCs. The formula was simple: gain more users, gain the attention of the bigger VCs that would fund you well. The startup founders did just that; they went after the "critical mass" of users to help kick off the success of their creation. With this, the flow of VC money poured into Asia and even gave rise to VCs grown from within Asia. In an article[1] written by the *Nikkei Asian Review* in October of 2019, the numbers for Asian-focused VC funds sat at 323 billion US $ under management, while the numbers for US-focused VC funds were at 397 billion US $. The numbers were taken at the end of 2018, and the article projected an eventual overshadowing of US VC funds by the ones in Asia.

So, what does this all mean? Well, the currency of startups these days is not actual currency. No. The majority of these startups are not even breaking even. The currency of startups in present day is users – users, glorious users. Users with their personal information, credit card numbers, addresses, email addresses, and unencrypted passwords or password hashes, it's quite the buffet for anyone that wishes to feast. The bigger the funding and the company, the more upmarket your buffet. I would even go so far as to say if you are tracking your attacks on your infrastructure regularly and see a rise to a new, higher norm, chances are you're making it big in the startup industry. Pat yourself on the back and go get yourself a CISO if you haven't one already.

[1] https://asia.nikkei.com/Business/Business-trends/Asia-set-to-eclipse-US-as-world-s-venture-capital-powerhouse

Between Two Books

A lot has transpired between the two editions of this book. Startups were formed, users were gained, users were hacked, and users were lost. Let's take a quick look at the security landscape between the years of 2013 and 2019. One thing that you may notice when you go back looking at breaches is that apart from news of Android malware, everything else seems like a hacking attempt. Let's look at the malware first.

What Is Malware?

I will use the description that I used in my first edition here. It hasn't changed:

> *Malware is defined as any piece of malicious software that lives on a user's computer or smartphone whose sole task is to cause destruction of data, steal personal information, or gain access to system resources in order to gain full control of the device it is on. Malware is written with the sole intent of causing harm; and usually, malware authors will write the malware to target a specific weakness in an operating system or platform. Often, the malware author will want to maximize the spread of her malware and will look to implement a mechanism where his software can copy itself to other, similar devices.*

Now, previously, I used the word "lives" on a user's computer or smartphone. Let's dissect how the malware gets to live on the device in the first place. Obviously before the malware starts living on the phone, it has to be introduced. To explore further, let's look at a piece of Android malware that has managed to steal over a million Google accounts. It's named Gooligan. Gooligan was first discovered as a mobile malware campaign that was not as malicious as it became.

Researchers from Check Point discovered the malware that came bundled with an application called SnapPea as shown in Figure 1-1.

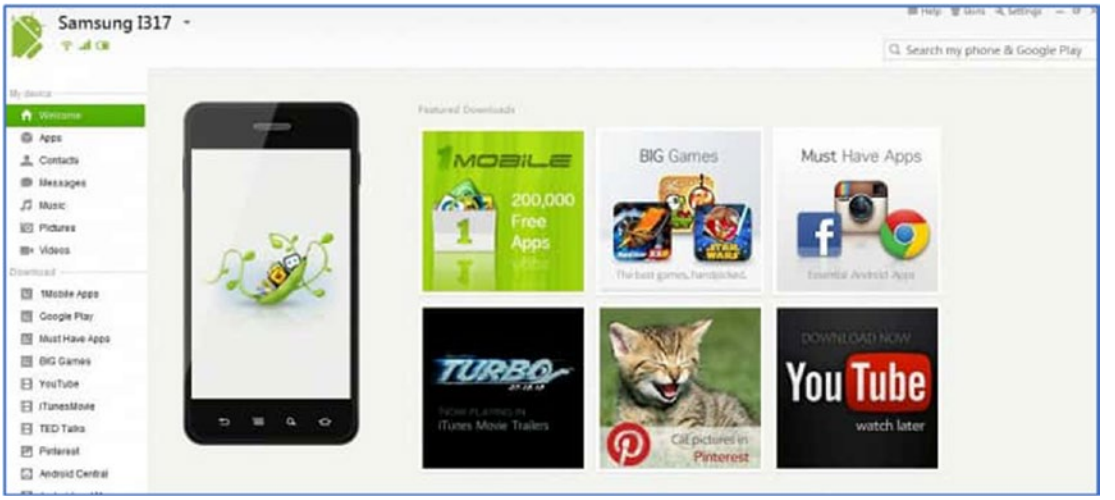


Figure 1-1. *The user interface for SnapPea*

Now SnapPea, it was found, would try to root your Android device with 12 different exploits. Check Point has remarked that it was the most amount of Android root attempts that they had witnessed in the wild. Two of the exploits were found to be the vRoot [CVE-2013-6282] exploit and the TowelRoot [CVE-2014-3153] exploit. These two are the only exploits that have CVE IDs in the database. When the rooting of the device was complete, the malware would install several other malicious apps that would further install subsequent ad-laden apps. The malware would then connect to a central server and await further instructions. It seemed like the primary purpose of this malware was to install other adware apps that made all infected devices connect to advertising sites and make it appear like legitimate users were viewing the ads. More views meant more payment for the app owners, and so greater infections yielded greater profits.

One thing interesting about the infection mechanism is that if SnapPea was installed from the Google Play Store, then there would be no infection. The app had to be downloaded from other sources that were most likely less stringent in how they checked the apps that they published. Also of note is that the user had to install the PC version of SnapPea and then connect his Android device to his PC in order to kick off the infection process. A blog post from Check Point shows the infection flow as depicted in Figure 1-2.

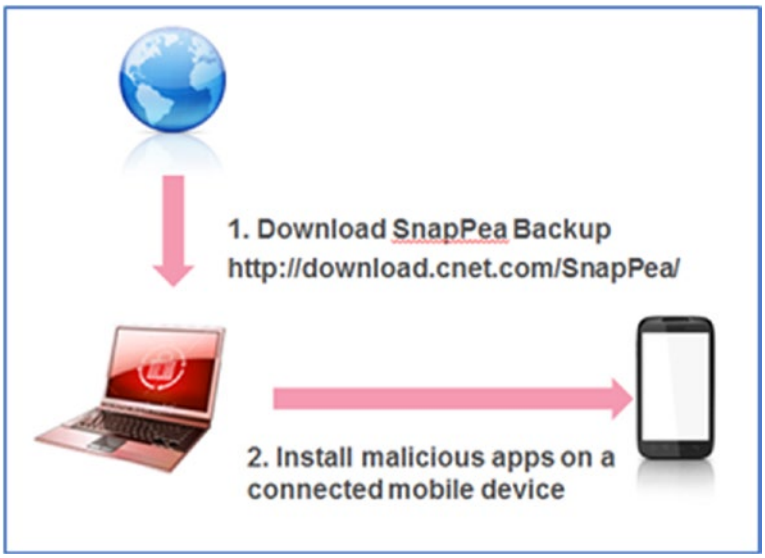


Figure 1-2. *How the malware made its way onto a device*

Subsequent iterations of this malware, which then became known as Gooligan, used similar rooting mechanisms, but had far more sinister goals in mind. After the original malware authors released their adware/malware campaign with SnapPea in 2015, they retreated and disappeared not to be heard from again until June of 2016 with Gooligan. Once again, apps that were available on third-party app stores carried the malicious payload. The apps were marketed as free versions of popular paid apps. So essentially, the draw this time was that you're getting a free version of an app which you would normally have to pay for – piracy.

Once installed, the app tried to root the device that it ran on. Then, it would tell a central command and control server information about the device and the fact that it was rooted. Then it would download a whole slew of rootkits and proceed to steal accounts and even authentication tokens for Google Photos, Play, Drive, Docs, and Gmail. A summarized flow of how Gooligan worked was found on the Check Point blog post[2] and was reproduced here in Figure 1-3. Gooligan left a trail of compromised accounts in its wake, and the number of accounts compromised went up from 400,000 in September of 2016 to 1 million by November 2016. In addition to stealing Google accounts, Gooligan would also download, install, and then review other seemingly ordinary apps and would leave a positive review on the Google Play Store for those apps.

[2] https://blog.checkpoint.com/2016/11/30/1-million-google-accounts-breached-gooligan/

The review text was received from the command and control server. Think of it this way, in November of 2016, there were approximately 1 million Android phones, downloading possibly useless apps from Google Play, installing them, and leaving positive reviews. It is highly likely that the Gooligan authors advertised a service in which app authors can get five-star reviews for a fee – similar to how you can find a service that guarantees to increase the count of your Twitter or Facebook followers.

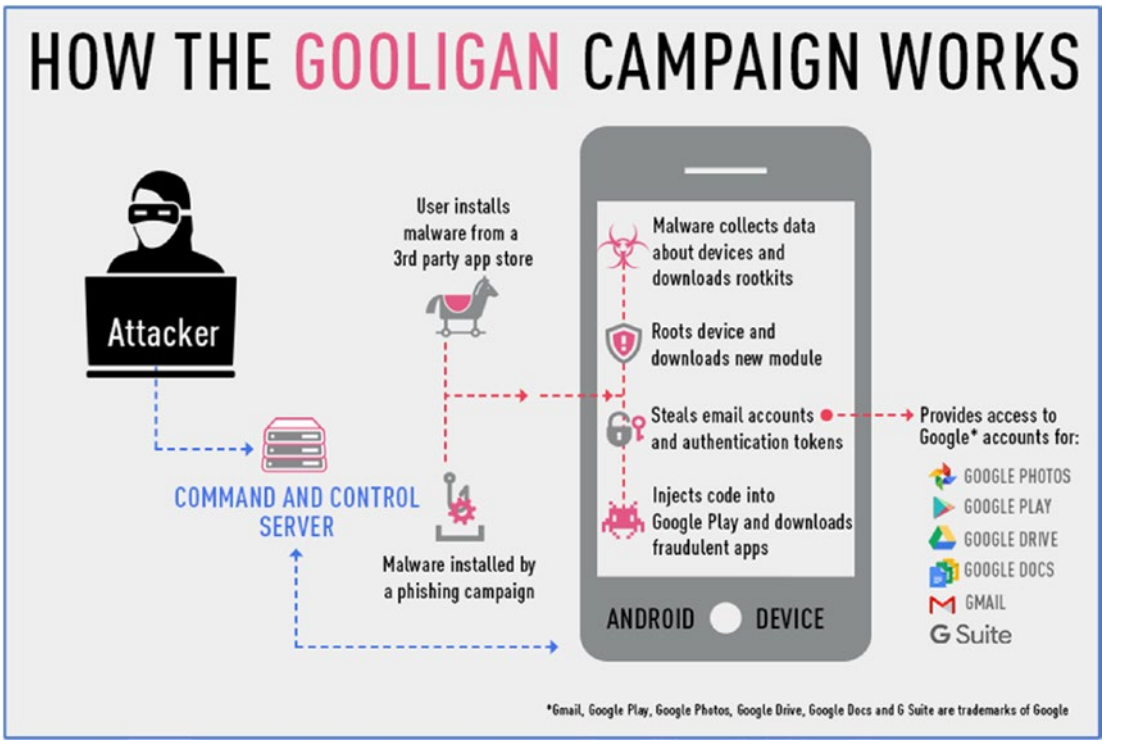


Figure 1-3. *How Gooligan worked*

Launching Attacks via Phones

Barring a situation similar to Gooligan earlier, it is important to keep in mind that attacking a phone gives you far less reward for your effort. The logistics of attacking a phone directly in that manner are beyond the reach of the solo hacker. Typically, organized malware attacks such as Gooligan are the work of larger, organized groups.

While the details of a lot of breaches still remain well hidden, my theory on how such large numbers of data records get stolen is that hackers launch their attacks from a semitrusted source. Bear with me on this one. If you consider the company that

builds an application of some sort from scratch, they will almost always have to build or integrate a back end with an API layer. They then build the mobile app to communicate with this API layer. Now an interesting thing happens that I have seen among teams working on these projects. The fact that both the back end and the mobile app are built by the same team gives a heightened sense of trust to the requests originating from the app side. The idea that all the data comes from the app that was also built in-house means that a slightly more relaxed security posture can be maintained at the back end. Let's analyze this first. Consider how the back-end developer may think about it:

- My API URL is an unknown, unpublished one
- My API can only be accessed by our mobile app that was built in-house

These assumptions are fairly normal ones to have in general if you have not had to think about the security of the overall product. Lucky is the paranoid developer that considers his code unleashed into hostile waters the minute he publishes it. The two preceding assumptions usually will lead to more lax security that can take on the form of sending too much data between phone and server, relying on the phone to carry the load of authentication, offloading some processing onto the phone, or storing sensitive data on the phone. The assumptions are wrong. Imagining for even a second that your data in transit between the app and the back end is safe from prying eyes is an illusion. I will show you later on in this book how to look through the data traversing between phone and server.

Note You may have seen me use the app and product earlier. By app, I mean just the application that is running on your smartphone. By product, I mean both the app and the back-end component which contains the API that the app will talk to.

OK, let's keep in mind that some developers may make the assumptions listed earlier and proceed to build and launch a product. We now have a product that makes assumptions about security. The product is released and the public loves it. Your startup begins to build a group of loyal users. Joy of joys you even begin to see repeat users when you do your cohort analysis! You begin to attract the attention of investors, and then the media starts writing about you. Let's hit the pause button here. Similar to the Hollywood movie effects, our actors all come to a halt with a sound that a record player makes when you physically stop the record with your finger.

What you have here is a pivotal moment when things can go colossally wrong, or you hire that CISO and they yell at you for getting this far without paying attention to security. Since we're tracking the security incidents between 2013 and 2019, let's go with colossally wrong. Let's unpause our scene.

Because of all the attention you have been getting from users, investors, and media, you also begin to attract the attention of bored or motivated hackers. A bored hacker who has a little too much time on his hand scrolls down past the news article about your product, then scrolls back up again. He thinks, "Hmm, I wonder what those guys are sending back and forth between back end and the app…." He downloads your app onto his rooted Android phone, plugs it into his laptop, then fires up his favorite set of tools on that laptop, and within minutes he begins to see all the traffic you send back and forth. He smiles, "Someone decided to send phone and credit card numbers in requests that only needed to send a name. Oh, that was particularly lazy of you to not filter out those bits of data." His smile widens as he whispers, "I have you now." End scene.

My gigantic ego would immediately associate myself with the superior skilled, elite hacker in the preceding story, if not for one thing: that was a true story of me when I tested an app in September of 2019. Yes, yes. As you will soon see, I have a flare for the dramatic. If you ask my daughter, she would even say melodramatic. Moving on to my theory, I firmly believe that a major portion of breaches involving a loss of large quantities of personal data stems from a weakness found in an app – a weakness similar to what we discussed earlier. The underlying assumption that all is well in the security of the product ecosystem can give rise to some stressful times that can make or break your business.

So, what truly happened between the first and the second edition of this book? Not a whole lot of change sadly. I've been testing app security as recently as two weeks before I started writing this book, and I still find the same attack vectors that I found over six years ago. Yes, this is an incredible unhelpful thing to say and probably goes on to even alienate some of the developers out there. I'm from the tough love school, and so, I feel that the best way to think about security, especially in a startup worth millions of dollars, is that you only get one chance. This is another reason why I am taking a crack at writing another book, so that we can all go through some of my experiences together and try to come out of it with better security. Right? Great! Let's now move on to some of that journey I've been promising you earlier in this chapter. Let's first start with some of my lows.

Hello, I'm Your CTO

As I am apt to tell whoever I am speaking with or addressing, I was extremely fortunate enough to join a startup in Southeast Asia, specifically Indonesia, called Gojek. Now if you haven't heard of Gojek, that's fine. Let me give you a brief idea of what we set out to do. I used to tell folks that were unfamiliar with Gojek that we were an Uber for the two wheelers aka motorbikes. To address your still burning questions, we must turn to the transportation infrastructure in Indonesia. If you ask an average Indonesian living in the capital city of Jakarta, they would say that the transportation system, regardless of what mode of transport you took, involved sitting stationary for large chunks of time. The word in Indonesian is called *macet* (mah-chet) which means "traffic jam." Macet was so much a part of the vernacular that a group of people you were meeting would nod sympathetically if you walked in 20 minutes late. Even if you had an actual legitimate reason or emergency for being late that didn't involve a traffic jam, your business counterpart would look up at your disheveled appearance and offer up a polite "Macet?" before you would start your meeting. Now because of macet, four wheelers aka cars would stand no chance in making up any lost time between your two meetings for the day. What emerged was an informal group of motorbike riders that thought, "Hang on, we can zip through macet and make some cash ferrying people who were late between their two meetings for the day." Essentially was born the culture of the motorbike taxi. It isn't entirely new and is usually found in cities that were developing fast and didn't have the infrastructure to cope. Thailand has one called the *motorcy*. In Indonesia, they called themselves *ojek*. Ojeks were great! They could get you back and forth fairly quickly but came with a slight problem: price haggling. If you wanted to get around, you would find an ojek asleep on his bike on a street corner, then tell him where you needed to go and why you were refusing to pay his ridiculous asking price. You can see where this is going, can't you?

Two friends of mine, Nadiem and Kevin, thought, "Hey, let's make an app for these ojeks! People don't like the fact that they have to haggle all the time so let's make that process simpler and build an app to hail and ride ojeks." Although Gojek was founded in 2010 as a rudimentary web page with a call center acting as an ojek dispatch, in 2015 they launched a mobile app for it. A rebirth if you will. The guys offered me to be one of the cofounders and picked me to be their first CTO, and you might think I jumped at the chance. Well, you would be wrong. I was skeptical to say the least when I heard the concept. I wondered where all the technology was that needed a CTO. But Nadiem, in his usual persuasive manner, asked me to try it for three months, and if I didn't like it,

then we could part ways. I ended up staying for almost four years. (One thing you should probably know is that a Gojek year was roughly equivalent to 3 normal years. I am deadly serious.) We went from a small startup of about 40 people in 2015 to one that is well over 3500 people today. An app that started with three core features then, transitioning to a super app with more than 20 features now. A company with minimal funding then, now valued at 10 billion dollars.

As the CTO, I was tasked with keeping the product running, building the product, building a team, and ensuring all our technology needs were met. To say that it was challenging would be an understatement – mostly because a lot of the things I had to do, I had to learn on the job. No prior role of mine prepared me for the complexity and stress that came with being the CTO of this new hot startup out of Southeast Asia. On most days, I felt like I had failed. We had so much growth, so few developers, and so many system failures that caused downtime. Our tech team was five people in those early days, and I would rack my brain trying to find and hire more developers while simultaneously helping with coding the back end, optimizing APIs and infrastructure maintenance. We went on like this for some time, and we slowly improved. Our product got better, we launched features fast, and we got funded. I was out of my element though. I was struggling because the role for me was one that I was diametrically opposed to: here I was building things when all I was used to doing was breaking things for over 15 years. I pushed on and learned a lot. I learned about building teams, about communication, and about accepting that good enough is good for now. I had never seen a scale or speed like that in my life, and it felt quite lonely because there was no one around that I could talk to about it. Yet I was happy, I was doing something I thoroughly enjoyed, I was challenged on a daily basis, and I didn't have time to think or take a break: bliss.

We were chugging along as usual balancing the product features, performance, and growth every day for roughly one year when I came to the realization that I was probably in over my head. While I knew that I could handle the role of CTO given enough time to catch up, this particular business was ruthless and unforgiving in the time that it gave you. I also knew that with our continued trajectory, we would desperately need to get a handle on security. Early on I had started to build a small security team that would sit there and hack our product to uncover any security flaws that could affect our users' privacy or affect the business negatively, but I knew that I would have to increase the size of this team and provide leadership in that area. With that, I had separate conversations with both Nadiem and Kevin and transitioned my role to be the CISO.

This was a bit of a low point in my time at Gojek because I was in a learning mode and doing something I had not done before, and being challenged at it was immensely satisfying. But in reality, there was a desperate need as we grew to make sure that security was tight, and I knew, given my background, I had to step up and deliver for the good of the company. As time passed, we even coined the phrase to signify important moments like this: "It's not about you."

The next low point was some time after I had transitioned from the CTO role to the CISO role; we received several emails from both well-meaning individuals and glory and profit-seeking individuals alike that told us they had successfully hacked our product and gave details on what they had uncovered. To strengthen my theory that I mentioned earlier, it turned out that they had breached our API back end by hacking our Android app. They discovered the communications patterns between the app and the back end and were able to compromise our product including collecting our customer data.
In the past, I have also been able to report flaws in products to companies including BlackBerry, and I always appreciated a company that reacted and responded fast. Still, nothing quite prepares you to receive an email like that – one that says that someone out there has managed to successfully compromise your product and offers you proof.
I felt my stomach lurch and a wave of nausea washed over me. Here I was, part of an organization that was on the receiving end of the hacking for a change; usually I was the one hacking others. Nevertheless, I took a deep breath, notified our leadership team, and took this to the tech team so that it could be rectified quickly.

Getting hacked is truly a terrible feeling. Having it unfold in public while hyped by the media for sensationalism makes that feeling worse. I empathize with anyone that has been through a scenario like that. It is a lesson that is swift and often one that you will not forget. It can make or break a company or a career. It leaves you feeling vulnerable and exposed. It is a feeling that I want to help you avoid. It is a mistake I want to prevent you from making. Therefore, I'm going to pack as much of this book as possible with real-world instances from which I learned some important lessons. Now there will most likely be different ways of handling a particular situation, but I will tell you how I did it.
In the first half of this book, I will discuss and document many of the areas you should be looking at as a builder of software when you are part of a startup. I will outline some techniques and tips that you can consider when you have to build something at scale. Your teams may be large and diverse, your communications may not be as streamlined as you would like, but I will talk about some practical mechanisms that you can adopt to improve your security as a builder. Not all of it will be a narrative as my stories in this

chapter; this **is** a technical book after all. Yet, I may occasionally wax poetic and regale you with my tales of the time I spent at this startup that is close to my heart, named Gojek. As for the second half of this book, I dedicate this to my people, the breakers…

Hello, I'm Your CISO

And I will help you to break apps so you can be better informed about how to fix them and avoid feeling what I felt when I got hacked.

A startup CISO's job is still fairly new. Acceptance has just begun because the number of articles I read on the topic is increasing steadily. Where to put your CISO, who should he report to, and what should his job description be are all discussed more frequently. Yet, the answers you will get will vary by and large by the person answering it and his agenda. I had plenty of discussions when I started in my role as CISO at Gojek – endless debate on whether I should report to the CTO or to the CFO or even the CEO. As you may have guessed, we had moved on a little from our startup roots at Gojek and were now emerging into a more structured operating model.

Let's tackle these questions in this section because I think they are important. Let's try to look at the role of the CISO from a few different perspectives. Let's start with the CEO and founder.

Reporting to the CEO

The founder and CEO of a startup does everything in the beginning. No task is too small; no job is beneath him. A founder and CEO will agonize over the longevity and survival of his idea and his company. How can I hire the best? How can I accelerate faster? How do I stay ahead of my competition? Where can I find money to pay my employees and keep the business running? These are some of the many questions that likely bounce around in his head. If we try to break down the role of a founder and CEO, it is one of removing friction at all costs. All his tasks and his purpose in the company are to remove friction and enable growth, sustainability, and survival of the idea, vision, and company. Thus, to report to a CEO, the CISO must be willing to be an enabler of growth, of technology, and of features. A CISO reporting to a CEO cannot be too restrictive, but at the same time, a CISO reporting to a CEO has to be an educator. He has to be someone that can demonstrably show the risks involved when setting out on a particular strategy. Communication is key in this relationship. More than that, it is important to be able to

understand your CEO and his goals so that you can effectively highlight the risks you may see on the path to these goals. For example, if your CEO is unaware of fraud that is taking place due to a flaw in the logic of your product, then learn to present the problem in terms that he would respond to. If he is a number or a data person, then quantify the fraud in the form of either a dollar value loss or a number of users lost datapoint. If he is more of a socially inclined person, then present how the fraud could cause a loss of productivity for the large number of users of your product. Your CEO is highly likely to be someone that has a bigger vision so understand that and try to adapt your narrative to show how what you propose will help drive toward that bigger goal.

Reporting to the CFO

This relationship, while rare, can still be one that you find yourself in. In this dynamic, the role of the CISO would most likely be weighted more onto the regulatory and compliance side of the spectrum. Technology security, such as red teaming and hacking your own product, may come second to ensuring that sufficient and sensible security policies and operating guidelines are in place. A CISO in this relationship may find that he has more leverage over individual teams or departments. The mantra of "do it this way or we are not in compliance" may follow this CISO and with good reason. Financial startups or startups that answer to regulators such as central banks have a strict set of guidelines on security that the startup must comply with. Often, noncompliance can lead to large fines or even a loss of an operating license that could bring the business to a crashing halt.

Reporting to the CTO

Don't.

I kid, I kid. This relationship is one that is mostly fraught with the greatest conflict of interest. The CTO's role in a startup is to deliver a feature-rich product that iterates fast and is performant. Finding a flaw or selecting a strategy that is more secure may directly affect the CTO's ability to do his job. We can all theorize about carving out a small team to fix any flaws discovered by the security team, but when it's crunch time and engineers are limited and the investors are breathing down your neck and the competition is seemingly moving faster, it becomes near impossible to slow down and address security flaws. This is a significant problem especially if the CTO isn't well versed in or does not come with a security mindset. Should a CISO find himself in this relationship, then

pre-agreeing with the entire leadership team about a structure of transparency should be something to consider. The worst position for a startup to be in is one where none of the security flaws discovered either in-house or externally are being brought to the leadership's attention. While there may be some short-term respite for the tech team, it rarely bodes well for the entire organization in the long run.

In my role as the CISO, I reported to Nadiem, the CEO. I structured my team in three broad areas of specialization: the Red Team, who would mount the offense on our product and infrastructure; the Blue Team, who would work to actively secure the infrastructure; and the Compliance Team, who would build, maintain, and disseminate policies and guidelines. The Compliance Team would also work together with the regulators on any licensing requirements that were necessary.

Reviewing What Gets Published

Being a fast-growing startup, one of the bigger challenges we faced as a security team was trying to figure out what products existed and which ones were launched newly. Essentially, this is inventorying, because you can't secure what you don't know about. Working closely with the infrastructure team, we took control of the process of adding a new set of APIs to our load balancer. This created a sort of gate that allowed us to question the requesting team whether they had conducted a preliminary penetration test on their system. If they hadn't, then we would not proceed with the release of the APIs until they did. Since we had announced this and continued to communicate this fact, it was rare that we encountered a team that requested an API addition having not completed a penetration test.

Another such "gate" scenario you can consider is to place a source code security review service as part of your software build process. Usually, tech teams will have build processes known as CI/CD or continuous integration/continuous delivery. Essentially, this boils down to having several teams contributing code that is merged to one main development branch and having shorter development cycles so that software can be deployed or delivered faster. Sometimes we would release software three or four times a day. Things like new features or bugfixes could go out very fast thanks to this process. One prerequisite to deliver the software was that the "build should pass." This means that prior to a release, all the source code that has been contributed is built. Once built, the software can be released. The build process will have a set of tests that it runs to ensure the software does what it is supposed to. For example, the component that added

a food item to your order was tested to see if when the item was added, the correct item was added, and that the correct quantity was added. If any of the tests would fail, especially on the critical tests, then the entire build process would stop with a failure notice. This meant that the software could not be released until the failures or bugs were addressed. By adding the source code review system to the path of the build process, your security team can ensure that any code committed to the main branch passes secure coding practices (at least as effective as the product used for source code review). If the source code reviewer found a flaw or an instance of insecure coding, then it would trigger a build failure. Thus, it would be possible to prevent insecure software from being released. Now keep in mind that this would just catch the low hanging fruit of the insecure source code. Although source code checkers are evolving, they are still not up to a level where they can prevent the majority of attacks.

Did I Just Waste My Time Reading All This?

I don't think so. Yes, it is a lot of narrative and very little to do with Android apps or anything deeply technical, but I think it sets the context for this book. I feel that by contextualizing some of the technical topics that come later in this book, it can add a little bit more value as to the "why" of doing things and doing things a certain way.

Startups, especially technology product startups, are here to stay. The mobile device is here to stay. It would be very interesting to get a glimpse into how all this will further evolve, but for now, looking at the tech and security through a startup lens adds a lot of value in my opinion. As the chapters unfold and go from defensive to offensive security, having this context in mind should serve you well. Just for the heck of it though, the offensive chapters should be fun to read without context either as I will write them as a handy reference guide that you can directly flip to, get setup, and go from there.

With all that being said, let's head over to our next chapter and review some of what we learned previously with regard to secure software development principles. See you there!

CHAPTER 2

Recap of Secure Development Principles

In Android coding with Kotlin or Java, you may have to go out of your way a little to write insecure code. I mean, there are the obvious ones like leaving your private key hardcoded, but in general you should do fine on the code front. Kotlin has made several improvements to Java, which was one of the only ways to write Android code natively in the past. Some notable, among the several, Kotlin improvements have been the addition of null safety and lesser quantity of code written. The null safety check in Kotlin means that if you were to access a variable that points to a null reference, then the system doesn't throw a `NullPointerException` or `NPE` like how Java used to. Kotlin reduces the amount of exception checking because it has no exception checking. This means that in cases where an interface throws a specific exception, you don't need to keep checking for exceptions each time you use one, for example, Java's `StringBuilder's append` method. Kotlin makes both try and throw expressions, so they can return a value which gives rise to code that looks like

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

or

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

But I'm not going to dwell at all on the language used to write Android apps in this chapter. Instead, I am going to look at instances where code written can pass on vulnerabilities to other parts of the system, like the back-end servers. I will also talk about some of the things you need to keep in mind with regard to your user's safety and privacy. So, let's get started. Let's look at privacy first.

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_2

Privacy

Data privacy has become a big deal lately. I mean, it always should have been, but the past few years have given us such wonderful headlines that privacy is almost always in the news. As a security practitioner, I'm thrilled that more people give a damn. As a security practitioner, I am also appalled at the amount of blatant violations of privacy still going on. Privacy is such a big deal that in May of 2018, the EU implemented GDPR or the General Data Protection Regulation. The law itself is fairly straightforward and logical, but interestingly, it allows for warnings and fines to the tune of up to 4% or 20 million Euros based on certain infringements. To date, 34 fines have been issued to various institutions including a hefty 50 million Euro fine to Google. Whether these have been paid or not are not part of this discussion, although I know Google was appealing their fine. The point is, however, to understand that we're getting serious about privacy.

If you're a developer that requests for and stores a user's private information, then you have a duty to keep that data private and safe. Depending on who you ask, the information one would consider private varies, but for a blanket definition, we typically define private information as information about a user that is generally not shared with the majority of the folks he meets. So, things like home address, gender, marital status, age, bank or credit card information, and phone number are usually considered the major components of information that a person would want to keep private. But let's analyze why first. I mean, why would you want to keep this information private? One can argue that by having that information about someone, there is precious little that can be done directly to affect him other than sending unsolicited pizza or a mob to his house or prank calling him until he disconnects his phone number. I'd like to pause here and talk about that mob I just mentioned. While I played it off as more or less innocent, there is one specific case where people have even died when mobs have been dispatched to their house. What I'm referring to here is the practice of swatting.

Swatting

Swatting takes place when someone malicious that knows your home address calls up emergency services and then deceives them into believing that there is a life-threatening emergency taking place at your house – typically, an armed person that has already hurt someone or a terrorist suspect. The resulting visit to your home by the SWAT or emergency response team can sometimes have catastrophic results. There was a case in 2017 where a 28-year-old father of two was shot by a police officer. The victim was not

even related to the group that initiated the swatting incident. He was an innocent victim that had the misfortune of having his address used for the purposes of winning a bet.

Let's go a little deeper into what happens when an emergency response team heads to a victim's house. They head there on full alert prepared to confront an armed and dangerous individual. The victim has no idea why someone has just broken into his home. He doesn't know if it is a friend or a foe. During heated times like that, sometimes a victim may resist or at least take on a defensive role. To the emergency response team, this may look like a challenge or threat, and if cooler heads don't prevail, then someone is taking a shot at the perceived aggressor. This is one way to look at why safeguarding your users' home addresses is important.

Aside from swatting, there is another key reason for keeping your user data private. And no, I'm not talking about the nasty headlines written about how badly you screwed up and allowed 100,000 user accounts to get leaked. I'm talking about social engineering. The more personal, private information that a social engineer has on a victim, the more plausible he can make his story. Humans are wired to trust. They are also wired to default to the truth in most cases. By using subtle cues of private information on a victim, it is a lot easier to push them into believing you and your story. "Oh this guy already knows about my information, so I guess he IS calling from that bank to help me with my bank account" would be a response from a hapless victim on the other end of the phone.

So, then, what's the secret to keeping a user's data protected? It's not really a secret. You have to protect your own infrastructure where the data is stored. One of the first steps I would advise would be to collect as little private information from a user as possible. It may be tempting to say "Let's just ask for this data now and figure out what to do with it later," but I think that is a bad idea. If you have a genuine need for the data, then ask for it – especially if it is something you can't do without, like a shipping address. The concept is that if you have the bare minimum amount of data, then you can rest easy knowing that the fallout from a breach isn't going to be that catastrophic, especially for the user.

The next piece of advice I would give is to separate the data into a so-called "Cold Storage" database. I've outlined how that may look like in Figure 2-1.

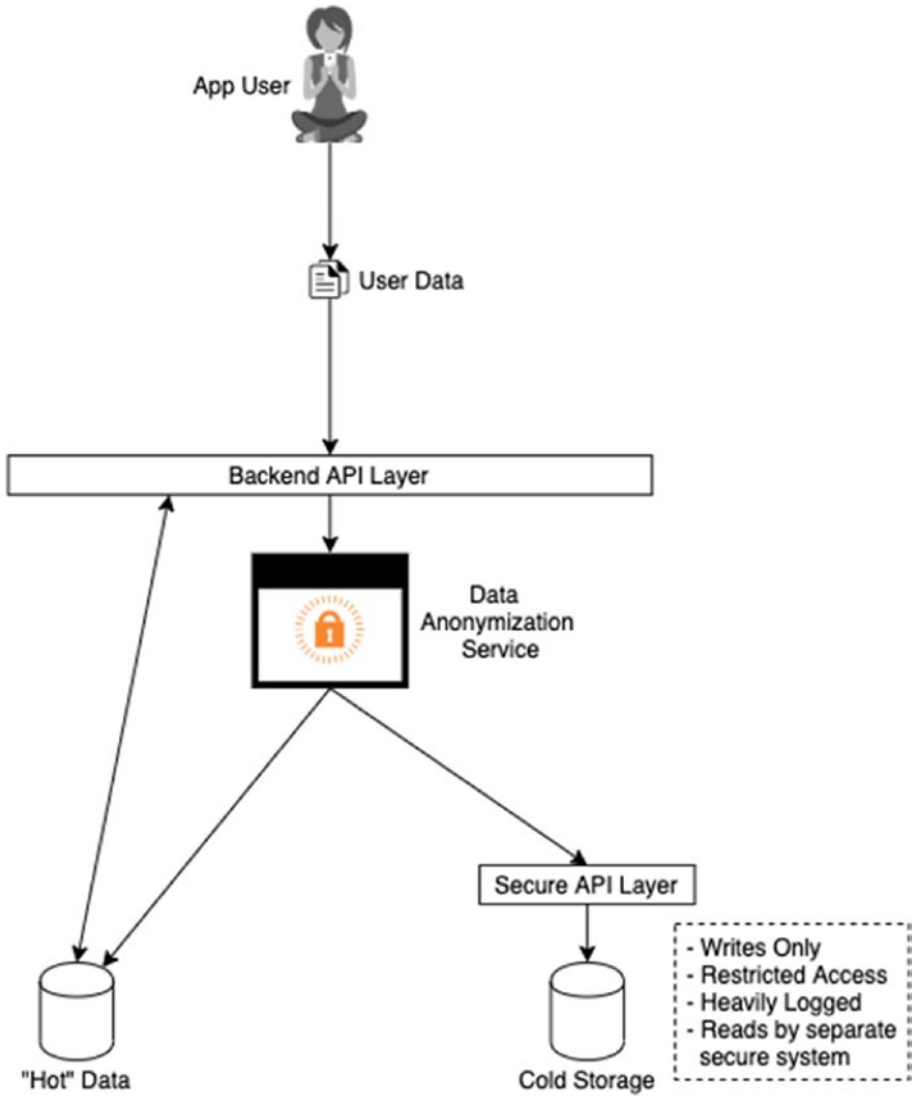


Figure 2-1. *Conceptual look at how the Cold Storage would work*

The concept here is that you split your data into two or more databases similar to how you would shard a database. Let's work through an example first and imagine an ecommerce platform. Let's look at registration. During registration, the user will complete his profile. Some data that we may want to collect could be as shown in Figure 2-2.

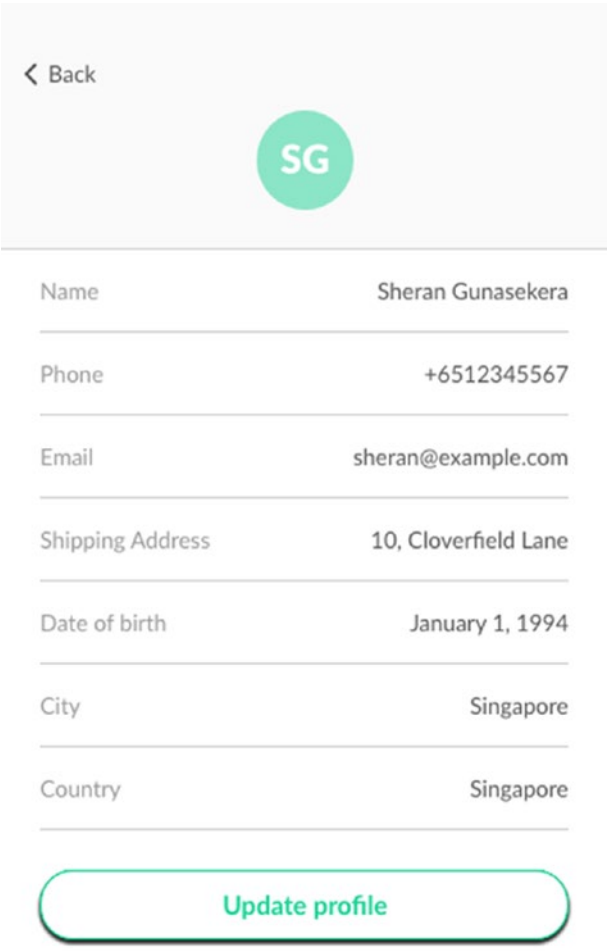


Figure 2-2. *Profile page when user registers*

Now when entering this data, the user can see fully what he inputs. What I recommend doing is as soon as he hits the Update Profile button, you take the following steps:

1. Generate a unique user id for him that you save in your main, "hot" database. This database is what gets used all the time.

2. Take all the private information and store it in a separate database that does not link to the main ecommerce site. This database is the more secure, "cold" database that holds private information. Typically, for the ecommerce site, it should be a write-only database. Information just goes in and does not come out to the ecommerce site.

3. With the information entered, take the private information that you just saved and render it illegible by using a basic algorithm such as blanking out all characters except the first character of each word. You can see an example of that in Figure 2-3. Once this is done, store it in the main database.

After this is done, when the user looks back at his profile, he will see it as unreadable. Since the user is familiar with his own data, he will be able to make sense of what it is. If an unauthorized user has access to his account or a data breach of the main database occurs, then the attacker will not see the full text of the private data.

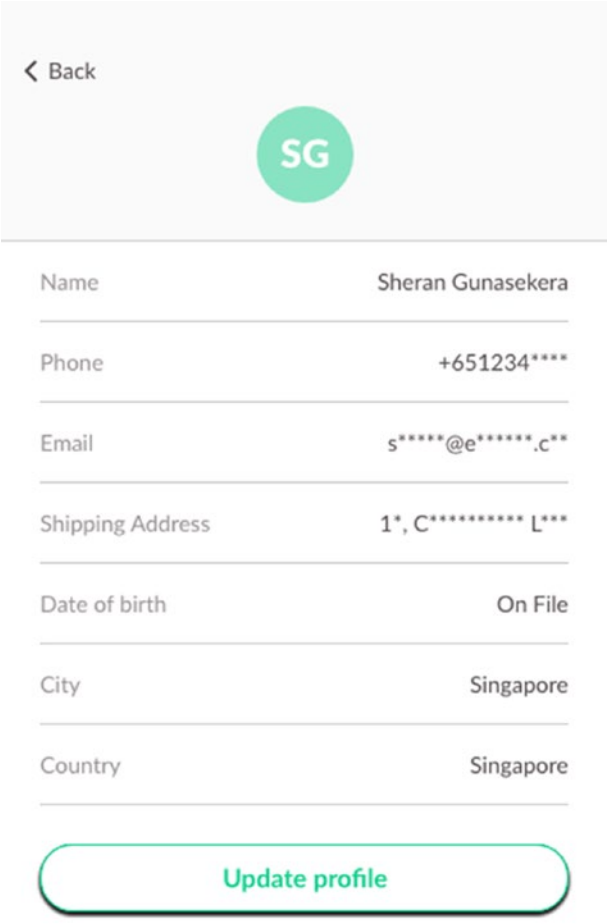


Figure 2-3. *The user profile when it is called up or read back by the user*

Another approach is to designate aliases to parts of the information stored. So, for his Shipping Address, you can get him to call it by an alias like "Home" or "Work." Then when it is read back, the address shows either Home or Work. Since the user knows what the home or work address is, he has all the information that he needs when he checks out. An attacker, however, will not be able to understand what the true address of the person is. All he will see is the word Home or Work.

The concept here is that the user will never need to be told what the information he entered is. He already knows that so there's no need to show him exactly what he entered. Instead, he will only want to modify or write to the profile data. By using a technique like this, you can greatly limit the exposure of your users in case a data breach takes place.

Data Security

Essentially, any data that an app may need, create, or modify has to be stored somewhere. Android has multiple disk partitions that house various parts of the operating system. We will look at the `/data` partition which contains the user's personal data. A few benefits to keeping the user data on a separate partition are that the rest of the operating system can be upgraded or recovered and even rewritten completely without concern for loss of user data. Next, the `/data` partition can be completely encrypted for security. Now encryption always comes with some overhead and can slow down performance. Since the other partitions don't contain any sensitive data, they can remain unencrypted, thus not adversely affecting the performance of the device. Within the `/data` directory is another /data directory. This is where all the installed applications reside. Here's what the directory looks like:

```
generic_x86:/data/data # ls -alrt|head
total 860
drwxrwx--x  37 system system 4096 2020-05-04 13:38 ..
drwx------   5 u0_a25 u0_a25 4096 2020-05-04 13:38 com.google.android.
             setupwizard
drwx------   6 u0_a67 u0_a67 4096 2020-05-04 13:38 com.google.android.
             deskclock
```

```
drwx------   7 u0_a22 u0_a22 4096 2020-05-04 13:38 com.google.android.apps.
             nexuslauncher
drwx------   4 u0_a32 u0_a32 4096 2020-05-04 13:38 org.chromium.webview_shell
drwx------   4 u0_a75 u0_a75 4096 2020-05-04 13:38 com.ustwo.lwp
drwx------   4 u0_a76 u0_a76 4096 2020-05-04 13:38 com.google.android.webview
drwx------   4 u0_a17 u0_a17 4096 2020-05-04 13:38 com.google.android.sdksetup
drwx------   4 u0_a51 u0_a51 4096 2020-05-04 13:38 com.google.android.
             printservice.recommendation
generic_x86:/data/data #
```

You will notice that each directory has its own user and group id. Essentially, Android will allow an app to write into its own space at will. Android apps are unable to write to any other location other than the `/data/data` directory.

Data Encryption

Android devices since 4.4 had the ability to fully encrypt the user partition. This meant that if your phone got into the wrong hands, then extraction of the data by means other than using your pin, password, or biometrics would not be possible without the decryption key. This should give you some peace of mind if you lose your phone, but of course, you absolutely must have a PIN or password lock enabled on your device. To enable full-disk encryption on your device, you can go to Settings ➤ Security ➤ Advanced ➤ Encryption. This screen for a Google Pixel 3 XL phone is shown in Figure 2-4.

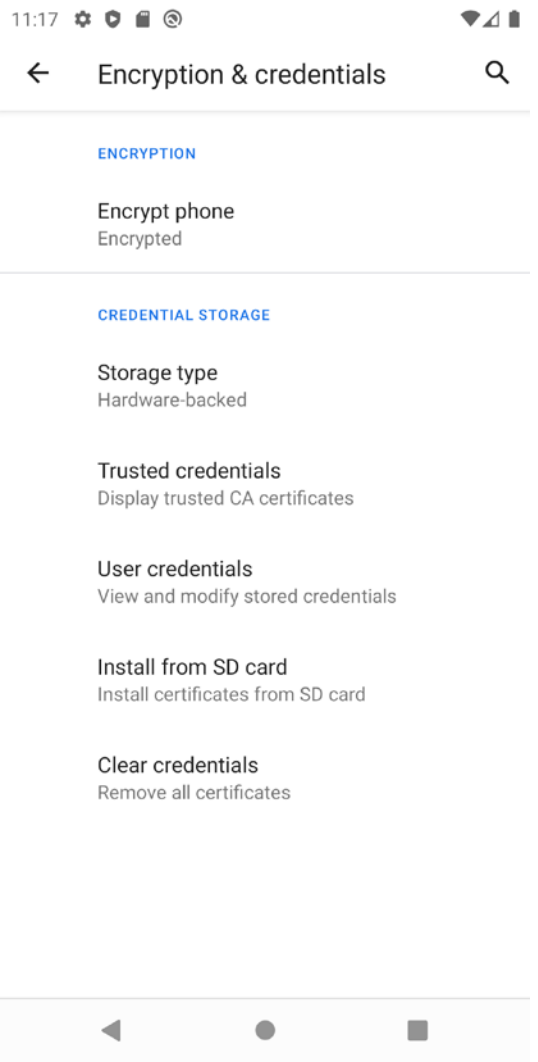


Figure 2-4. *Encryption settings on a Google Pixel 3 XL device*

While this security feature is great, you must also consider the data contained in your app. Again, similar to the privacy debate, I think it is important to only store data that you don't mind someone looking at on your device. Thus, store only nonsensitive, nonprivate data. If your app decides to store data in SQLite databases, then you would do well to consider the SQLCipher from Zetetic [`https://github.com/sqlcipher/android-database-sqlcipher`]. SQLCipher adds a layer of encryption to your databases that you use in your app and does so in a very transparent manner. Figures 2-5 and 2-6 show you what it looks like when SQLCipher encrypts a database.

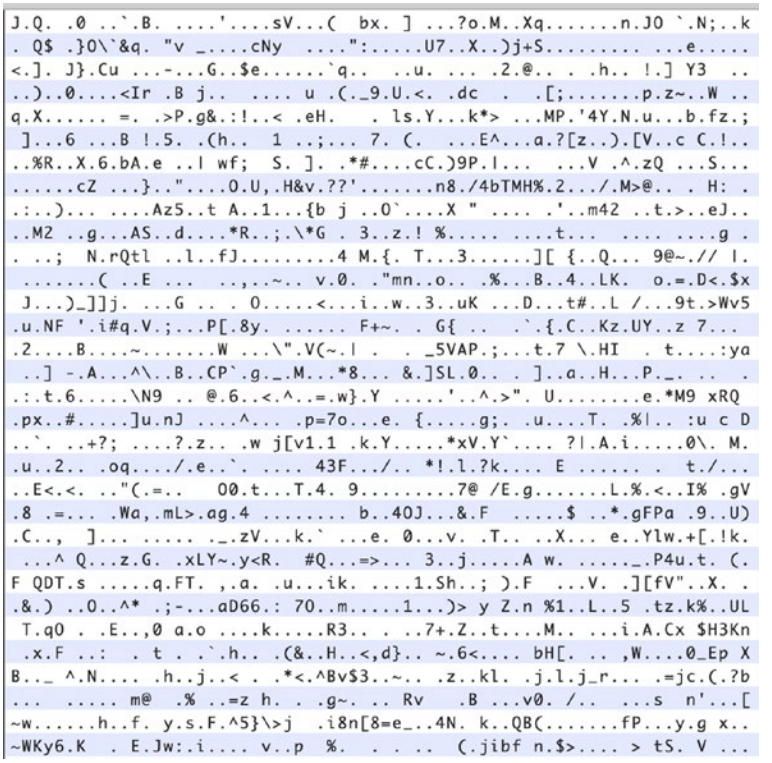


Figure 2-5. *What the BMW Connected App's database looks like in a hex editor*

```
→  bmw file *.db
ConnectedDB.db:                    data
applications.db:                   SQLite 3.x database, user version 615
google_app_measurement_local.db: SQLite 3.x database, user version 1,
→  bmw
```

Figure 2-6. *All the databases pulled out of the BMW Connected App*

The BMW Connected App makes use of SQLCipher to keep its data encrypted on the device. In Figure 2-6, we run the `file` command on the database files pulled from the BMW Connected App. You can clearly see how the non-encrypted databases were correctly identified as SQLite files, but the ConnectedDB.db file, which uses SQLCipher, shows up as plain data. That is because it is encrypted. Figure 2-5 shows a hex dump of the ConnectedDB.db file.

Zetetic kindly provides the Android version for free via their SQLCipher Community Edition found at the preceding URL. Setting up and using SQLCipher is fairly straightforward. You add the following dependencies into your `app/build.gradle` file:

```
implementation 'net.zetetic:android-database-sqlcipher:4.3.0@aar'
implementation "androidx.sqlite:sqlite:2.0.1"
```

After this, you can write code as you normally would if you wanted to use `android.database.sqlite.SQLiteDatabase`, except this time you would use `net.sqlcipher.database.SQLiteDatabase` instead.

Then, using the Kotlin language, you can write some demo code as follows:

```
package com.demo.sqlcipher

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import net.sqlcipher.database.SQLiteDatabase

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        SQLiteDatabase.loadLibs(this)
        val databaseFile = getDatabasePath("demo.db")
        if(databaseFile.exists()) databaseFile.delete()
        databaseFile.mkdirs()
        databaseFile.delete()
        val database = SQLiteDatabase.openOrCreateDatabase(databaseFile,
        "test123", null)
        database.execSQL("create table t1(a, b)")
        database.execSQL("insert into t1(a, b) values(?, ?)",
            arrayOf<Any>("one for the money", "two for the show")
        )
    }
}
```

Calling Up Sensitive Information

If there is a reason where you are unable to obscure the sensitive information before you present it to the user, then a good practice is to always prompt for user credentials before displaying it to them. Ideally, you want to use the device's PIN or password or even biometric authentication if available. Here is some Kotlin sample code from https://developer.android.com that shows an example usage of biometric authentication in your app.

```
private lateinit var executor: Executor
private lateinit var biometricPrompt: BiometricPrompt
private lateinit var promptInfo: BiometricPrompt.PromptInfo

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_login)
    executor = ContextCompat.getMainExecutor(this)
    biometricPrompt = BiometricPrompt(this, executor,
            object : BiometricPrompt.AuthenticationCallback() {
        override fun onAuthenticationError(errorCode: Int,
                errString: CharSequence) {
            super.onAuthenticationError(errorCode, errString)
            Toast.makeText(applicationContext,
                "Authentication error: $errString", Toast.LENGTH_SHORT)
                .show()
        }

        override fun onAuthenticationSucceeded(
                result: BiometricPrompt.AuthenticationResult) {
            super.onAuthenticationSucceeded(result)
            Toast.makeText(applicationContext,
                "Authentication succeeded!", Toast.LENGTH_SHORT)
                .show()
        }

        override fun onAuthenticationFailed() {
            super.onAuthenticationFailed()
            Toast.makeText(applicationContext, "Authentication failed",
                Toast.LENGTH_SHORT)
```

```
                .show()
        }
    })

    promptInfo = BiometricPrompt.PromptInfo.Builder()
            .setTitle("Biometric login for my app")
            .setSubtitle("Log in using your biometric credential")
            .setNegativeButtonText("Use account password")
            .build()

   val biometricLoginButton =
            findViewById<Button>(R.id.biometric_login)
    biometricLoginButton.setOnClickListener {
        biometricPrompt.authenticate(promptInfo)
    }
}
```

Network Security

By now I think every developer should know to always send their network traffic over SSL or TLS. Network traffic sent over plain HTTP can be very easily sniffed by using a tool like Burp Proxy. All you would have to do is to fire up Burp Proxy, which listens on port 8080, then add a proxy to your Android's Wi-Fi connection to route all traffic to Burp Proxy. Of course, both devices have to be on the same network. These days, even plain old HTTPS is not enough either. You can easily circumvent the hassles of sniffing TLS traffic by installing Burp Suite's TLS Certificate which the app ends up recognizing. After this, it has no reservations about sending Burp Proxy all its TLS traffic as well. What's the solution then? The main one I wanted to cover here is known as SSL Pinning.

SSL Pinning is a technique you can use to ensure that your app only speaks to a specific server that it trusts. You do this by telling your app to only trust TLS Server Certificates with a specific fingerprint. You can make the trust levels as broad as you want to or as narrow as you want to. You can, in fact, only make your app trust one host name and one TLS Certificate. A flowchart of how SSL Pinning works in concept is shown in Figure 2-7.

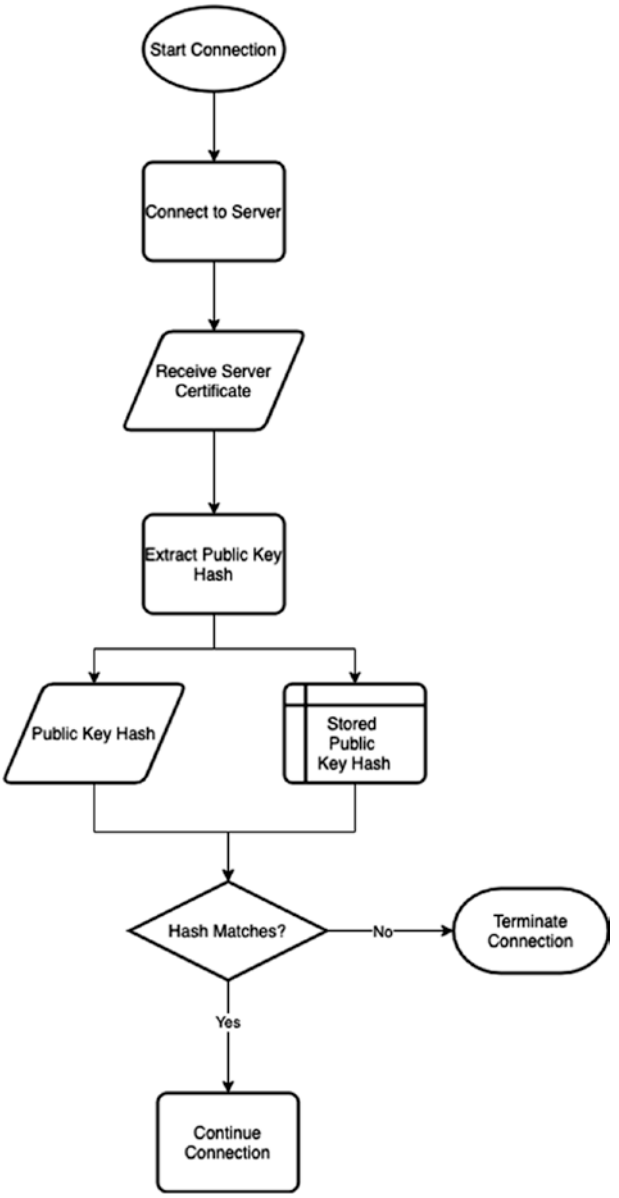


Figure 2-7. *Flowchart of how SSL Pinning works*

We will cover SSL Pinning and its related security in depth in Chapter 9. A snippet of code that shows one of the ways in which SSL Pinning can be implemented in Android is shown as follows:

```
01: package com.redteamlife.aas2.aas2client
02:
03: import android.os.AsyncTask
04: import android.util.Log
05:
06: import org.json.JSONObject
07: import java.io.*
08: import java.net.URL
09: import java.security.KeyStore
10: import java.security.cert.CertificateFactory
11: import javax.net.ssl.*
12: import java.security.cert.X509Certificate
13:
14: class NetworkAsyncTask(activity: MainActivity): AsyncTask<String, Void,
    String>(){
15:
16:     private val mActivity = activity
17:
18:     override fun doInBackground(vararg params: String?): String? {
19:         val cf: CertificateFactory = CertificateFactory.
          getInstance("X.509")
20:
21:         val caInput: InputStream = BufferedInputStream(mActivity.
          resources.openRawResource(R.raw.cert))
22:         val ca: X509Certificate = caInput.use {
23:             cf.generateCertificate(it) as X509Certificate
24:         }
25:
26:         val keyStoreType = KeyStore.getDefaultType()
27:         val keyStore = KeyStore.getInstance(keyStoreType).apply {
28:             load(null, null)
29:             setCertificateEntry("ca", ca)
30:         }
31:
```

```
32:         val tmfAlgorithm: String = TrustManagerFactory.
            getDefaultAlgorithm()
33:         val tmf: TrustManagerFactory = TrustManagerFactory.
            getInstance(tmfAlgorithm).apply {
34:             init(keyStore)
35:         }
36:
37:         val context: SSLContext = SSLContext.getInstance("TLS").apply {
38:             init(null, tmf.trustManagers, null)
39:         }
40:
41:         var connection: HttpsURLConnection? = null
42:         return try{
43:             connection = (URL(params[0])?.openConnection() as?
                HttpsURLConnection)
44:             connection?.requestMethod = "POST"
45:             connection?.doOutput = true
46:             connection?.doInput = true
47:             connection?.setRequestProperty("Content-Type","application/
                json")
48:             connection?.sslSocketFactory = context.socketFactory
49:             connection?.connect()
50:
51:             val message : JSONObject = JSONObject()
52:             message.put("code",params[1])
53:             val outputWriter = OutputStreamWriter(connection?.
                outputStream)
54:             outputWriter.write(message.toString())
55:             outputWriter.flush()
56:             if (connection?.responseCode == 200){
57:                 val inputStream = InputStreamReader(connection?.
                    inputStream)
58:                 val body = JSONObject(inputStream.readText())
59:                 return body.getString("message")
60:             } else{
```

```
61:                 return "error"
62:             }
63:         } finally {
64:             connection?.disconnect()
65:         }
66:     }
67:
68:     override fun onPostExecute(result: String){
69:         mActivity.setText(result)
70:     }
71:
72:
73: }
```

CHAPTER 3

App Licensing and SafetyNet

In this chapter, I want to look at how you can understand the integrity of your app and the integrity of the device that your app runs on. Let's take those two topics and drill down a bit further.

On the app side, when you write and publish an app, you want to ensure that the app that is running on the end user's device is indeed your app. In cases of commercial apps or apps that have a commercial component to it within the app, for example, buying additional features, you will want to make sure that the code being run and the back end that it is talking to know that it was built by you or your team. The reason you would want this check in place is if an attacker reverse engineers and modifies your app. For instance, an attacker can try to "purchase" or obtain certain paid content or features from your app by reverse engineering and patching your app. For instance, he can bypass a trial period check by patching the check to see if the trial period has expired. To do this, he would have to reverse engineer the app, edit the code, rebuild the app, sign it with his own key, and install it on his own device. Your app, without having a means to verify where it was actually installed from, will continue to work. App Licensing aims to correct this attack vector by working together with Google Play's licensing server to make sure the app was legitimately downloaded or purchased through the Play Store. Any modified, side-loaded apps can be filtered out and disabled by this mechanism. I cover how to get App Licensing work in the first edition of this book in Chapter 9 – "Publishing and Selling Your Apps." The concept itself remains the same; however, for an updated set of documentation, please see here: `https://developer.android.com/google/play/licensing`.

On the topic of ensuring the device that your app runs on, there is Android SafetyNet. On Google's documentation for SafetyNet, we see the description that SafetyNet helps protect your app against security threats such as tampered devices, bad URLs, harmful

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_3

apps, and fake users. Each of these topics is covered here [https://developer.android.com/training/safetynet], though I want to take a closer look at how to protect apps from being run in tampered devices. I will take a closer look at how to implement SafetyNet and look at some examples in practice. To do this, we will write an app that makes use of SafetyNet's Attestation API and see how that works. Before that, let's take a look at how the Attestation API works. Take a look at Figure 3-1.

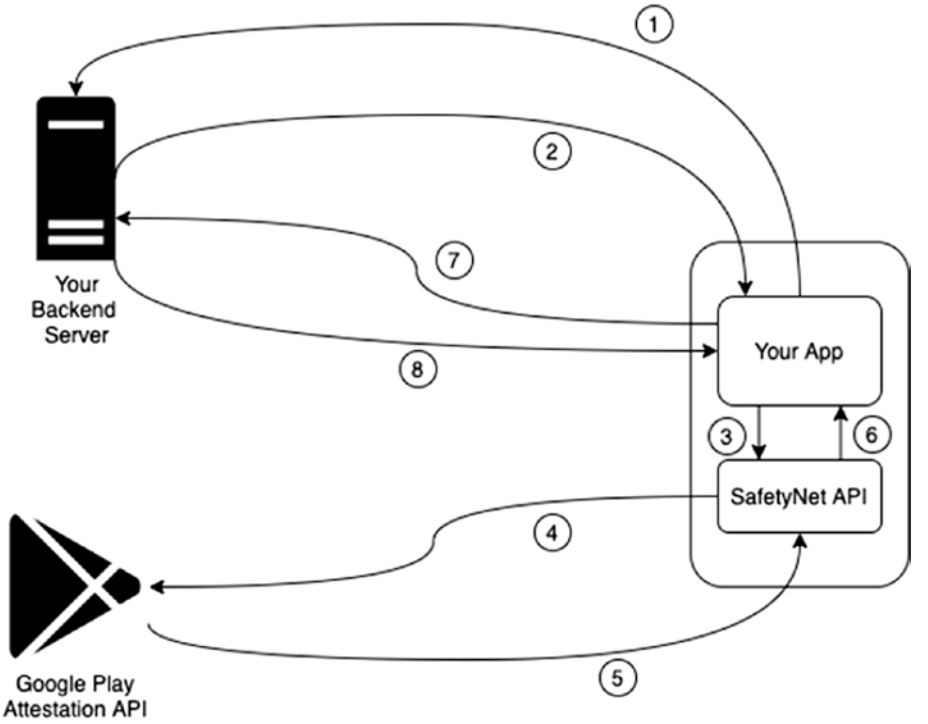


Figure 3-1. *The steps taken for SafetyNet Attestation*

Let's look at each of the steps:

1. First, whenever a user of the app requests a feature or an action, the app will make a request to the back end to request a nonce.
2. The server responds with the nonce. This nonce is important for the lifetime of the entire transaction, and it prevents replay attacks. The nonce should be completely random and used only once.
3. The app calls the SafetyNet Attestation API in Google SafetyNet services on the device.

4. The SafetyNet Attestation API will evaluate the device state and then send a request to the back-end server of the Google SafetyNet Attestation API to reply with a signed response giving back the results of the attestation.
5. Google's servers return a signed set of attestation results.
6. The SafetyNet services respond with the results to your app.
7. Using this set of results, your app calls your back end to fetch whatever user request there was.
8. Your back end decides whether or not to honor the request for services based on the results of the attestation. If the results show that the device has been tampered or has root, then you can choose not to interact with the device.

It is important to note the involvement of the back-end server in this whole transaction flow. Ultimately, it is the back-end server that enjoys a higher level of security and should be more trusted than getting results from the device. The device itself could be compromised and return a "Yup! I'm all safe over here" response, but that could entirely be fabricated. Hence, the need for the final validation from the server. Prior studies that have been done by independent companies have shown a very small percentage of developers or apps that make use of Google's SafetyNet. This is understandable when you consider the amount of extra work that you have to do as a developer to get this implemented. The ideal scenario is that you write this into some sort of helper library and have it available for all your in-house developers to use. You can choose to enforce this at build time by doing a static source code assessment and failing the build if you detect no use of SafetyNet in all critical requests to the back end. The requests to the SafetyNet API are expensive and will add overhead to your requests. Therefore, you may want to consider that as well. You could also choose to make a single recurring call for the duration of your app usage, but this may not always work out, especially if your app is not long running or gets used for a prolonged duration of time.

Now let's see how all this looks like in an app. For this demo, we will build both our app and our back end. Let's first define what we want to do. Let's say that we are a delivery bike rider and that we want to set our status as ready to accept deliveries from the back end. Let's build out our application first. Let's install Android Studio first so that we can use it to build our app. Visit `https://developer.android.com/studio` and click the Download Android Studio button. I am on a Mac, so these instructions will be

MacOS focused; where possible, I will try to give Windows equivalents. Read and agree to the terms of use for Android Studio and download the package. Once downloaded, double-click to open the disk image. Drag the Android Studio icon into the Applications folder to install. In Windows, when you are asked if you want to allow the app to make changes on your device, verify that it says Android Studio and click Yes. At the first screen, click Next. Then you're asked to pick components to install; leave the default ones and click Next. Then you're asked to pick an install location. For this, as well, leave the default and click Next. Lastly, click Install on the window that comes up and wait while the files are installed. When the installation is complete, click Next on the resulting window and then click Finish to start Android Studio.

When Android Studio starts up, it will ask you if you want to import settings from a previous installation. I choose "do not import" and click OK. Then you should be brought into an Android Studio setup wizard. Click Next and select a Standard install type. Then pick your screen theme colors, click Next, and verify the options to install. On OS X, click Finish and wait while the final setup is done. On Windows, click Next at the SDK Components Setup and then click Finish. After final setup is done, click Finish again and you should end up with a screen that looks like the one in Figure 3-2.

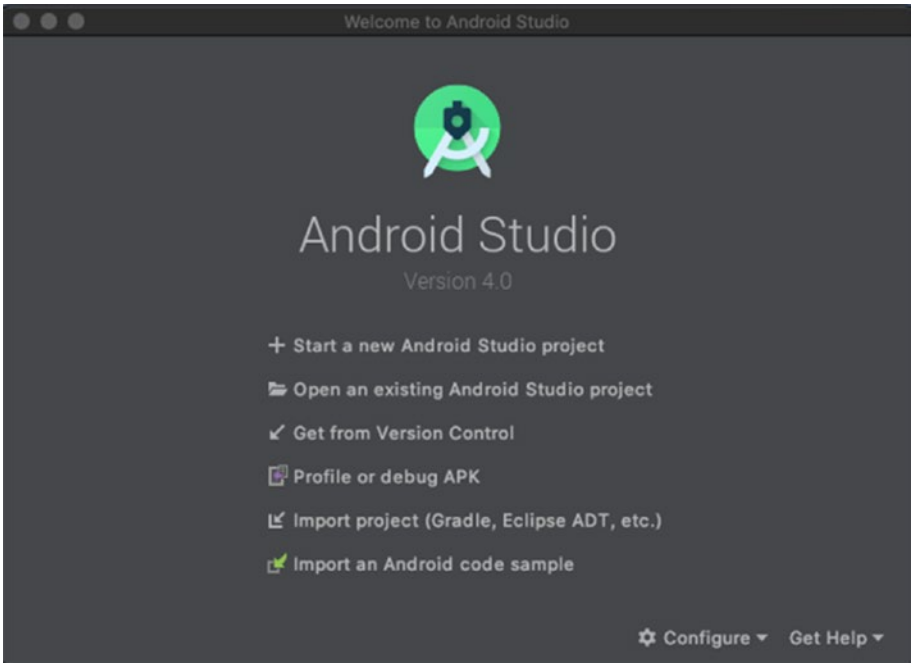


Figure 3-2. *Android Studio start screen on OS X*

Next, choose "Start a new Android Studio project" and then select Empty Activity from the resulting window. Next, we're going to name our project. Pick a package name and an application name, the location where you will save the project (I leave it default), and the minimum SDK level required to run your project. Android Studio helpfully shows you the approximate percentage of all Android devices your app will run on based on your minimum SDK selection. For this project, I am giving the package name `com.redteamlife.aas2` and the app name of `aas2attest`. The project configuration is shown in Figure 3-3.

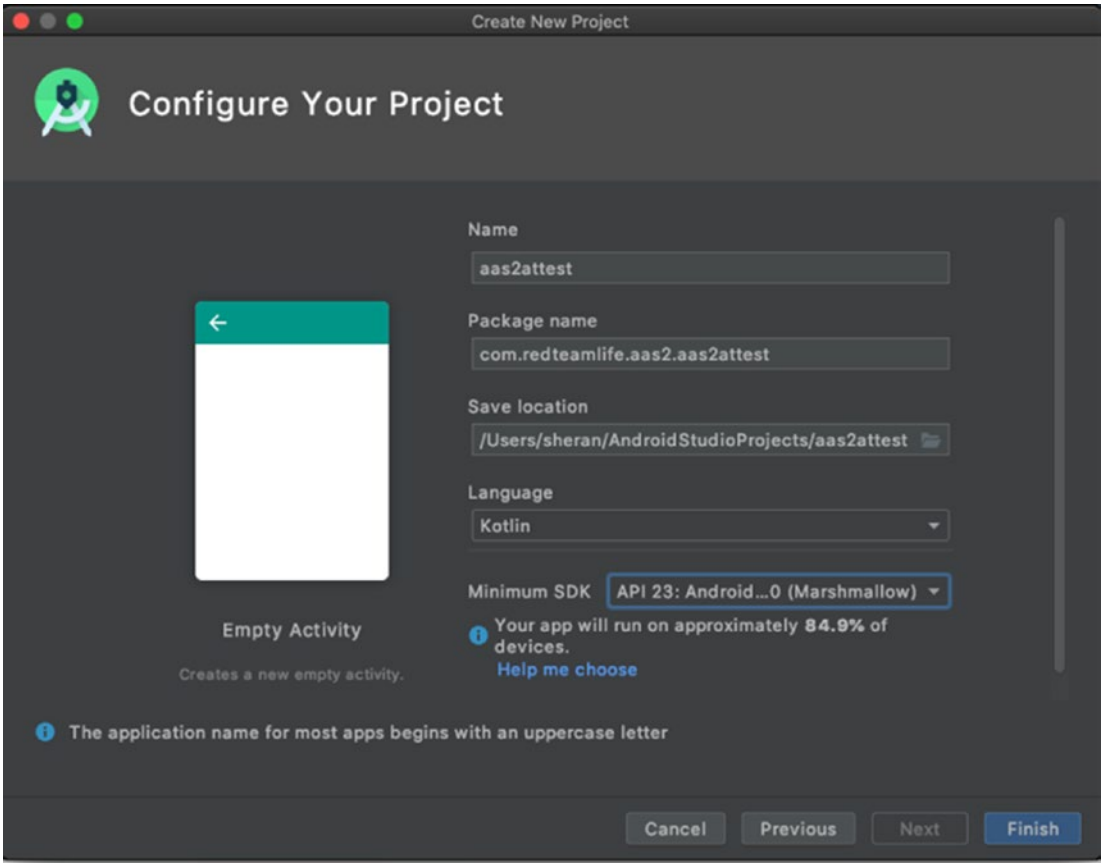


Figure 3-3. *Naming our new project*

I am including the full source code for you here, and you can also find both the client and the server source code at `https://github.com/sheran/aas2-ch03-android_attest_client` and `https://github.com/sheran/aas2-ch03-go_attest_backend`, respectively. I do, however, want to go over two of our classes which we use in the project and break down what they do:

Our MainActivity.kt file

```
01: package com.redteamlife.aas2.aas2attest
02:
03: import androidx.appcompat.app.AppCompatActivity
04: import android.os.Bundle
05: import android.util.Log
06: import android.widget.CompoundButton
07: import android.widget.Switch
08: import com.android.volley.toolbox.JsonObjectRequest
09: import com.android.volley.toolbox.RequestFuture
10: import com.android.volley.toolbox.Volley
11: import org.json.JSONObject
12: import java.util.concurrent.ExecutionException
13: import java.util.concurrent.TimeUnit
14: import java.util.concurrent.TimeoutException
15:
16: class MainActivity : AppCompatActivity() {
17:
18:     override fun onCreate(savedInstanceState: Bundle?) {
19:
20:         val TAG = "aas2attest"
21:
22:         super.onCreate(savedInstanceState)
23:         setContentView(R.layout.activity_main)
24:
25:         val attest = Attest(this)
26:
27:         val switch = findViewById<Switch>(R.id.switch1)
28:         switch.setOnCheckedChangeListener { buttonView, isChecked ->
29:             if (isChecked) {
30:                 switch.isEnabled = false
31:                 attest.getNonce()
32:             }
33:         }
```

```
34:     }
35: }
36:
```

The MainActivity file is not very busy as such. We have a simple switch widget that we have added to our Activity, and we check whether it is moved into the checked position or not in lines 28–29. If it is enabled, then we begin our process for attestation – lines 30–31.

Our Attest.kt file

```
01: package com.redteamlife.aas2.aas2attest
02:
03: import android.util.Log
04: import android.widget.Switch
05: import com.android.volley.Request
06: import com.android.volley.Response
07: import com.android.volley.toolbox.JsonObjectRequest
08: import com.google.android.gms.common.ConnectionResult
09: import com.google.android.gms.common.GoogleApiAvailability
10: import com.google.android.gms.safetynet.SafetyNet
11: import org.json.JSONObject
12:
13:
14: class Attest(activity: MainActivity){
15:
16:     private val mActivity = activity
17:     val TAG = "aas2attest"
18:     val main_url = "https://aas2.redteamlife.com:8443/"
19:
20:     fun getNonce(){
21:
22:         val queue = RQueue.getInstance(mActivity.applicationContext).
            requestQueue
23:         val url = main_url+"nonce"
24:         val jsonReq = JsonObjectRequest(Request.Method.
            GET,url,JSONObject(),Response.Listener {response ->
```

```
25:             requestAttest(response.getString("nonce"))
26:         },Response.ErrorListener {error ->
27:             Log.d(TAG,error.toString())
28:         })
29:         RQueue.getInstance(mActivity.applicationContext).
            addToRequestQueue(jsonReq)
30:     }
31:
32:     fun requestAttest(nonce:String){
33:         val API_KEY = "<insert _your_api_key_here>"
34:
35:         if (GoogleApiAvailability.getInstance().isGooglePlayServices
            Available(mActivity.applicationContext)
36:             == ConnectionResult.SUCCESS) {
37:             SafetyNet.getClient(mActivity.applicationContext).
                attest(nonce.toByteArray(), API_KEY)
38:                 .addOnSuccessListener { resp -> validate(resp.
                    jwsResult) }
39:                 .addOnFailureListener { err -> Log.d(TAG,err.
                    toString()) }
40:         } else {
41:             Log.d(TAG,"Install Play Services")
42:         }
43:     }
44:
45:
46:     fun validate(jwsResult: String){
47:         val queue = RQueue.getInstance(mActivity).requestQueue
48:         val url = main_url+"validate"
49:         val jsonObj = JSONObject()
50:         jsonObj.put("jws_result",jwsResult)
51:         jsonObj.put("action","change_state")
52:
53:         val sw = mActivity.findViewById<Switch>(R.id.switch1)
```

```
54:         val jsonReq = JsonObjectRequest(Request.Method.
            POST,url,jsonObj,Response.Listener {response ->
55:             sw.isChecked = response.getBoolean("validation")
56:             sw.isEnabled = true
57:         },Response.ErrorListener {error ->
58:             Log.d(TAG,error.toString())
59:         })
60:         RQueue.getInstance(mActivity.applicationContext).
            addToRequestQueue(jsonReq)
61:     }
62: }
```

In our Attest.kt file, we have three methods in lines 20, 32, and 46. The first method, `getNonce()`, does just that; it fetches a nonce from our back-end server. Once it gets a nonce, it calls the `requestAttest()` method and passes this nonce to that method. This is the method that then queries Android SafetyNet APIs. Behind the scenes, the Android SafetyNet library makes a call back to Google's APIs. Since you end up accessing Google's APIs for this, you will need to get your own API key in order to call this specific endpoint. I will show you the steps in getting your API key later on. After you receive the result of the call, you are then ready to decide whether to allow or disallow the specific request that was just made. The best way to do this is, once again, on the server side as opposed to the client side. The client side can easily be intercepted and altered, and that is why we should never fully trust the end user's device that he runs our apps on. That is why we have the `validate()` method. This last method will send the entire response from Android SafetyNet back to your own back-end server where you can decide whether or not to honor the request. We will talk about this more, but let's first do two things: build a back end and get an API key.

API Key

As mentioned before, Google requires you to register for an API key before you call the SafetyNet Attestation API. You will need to have a Gmail account or a G Suite account to do this, so have this handy. In your browser, visit this URL: https://console.developers.google.com/apis/library and sign in if required.

Then, in the *Search for APIs and Services* search bar, search for and select the Android Device Verification API module. Then, enable it by clicking the Enable button. Figure 3-4 shows what that looks like. Each API requires that it be tied into a Google Cloud Platform project, so if you don't already have a project, one named My First Project will automatically be created for you and the API enabled linked to this project.

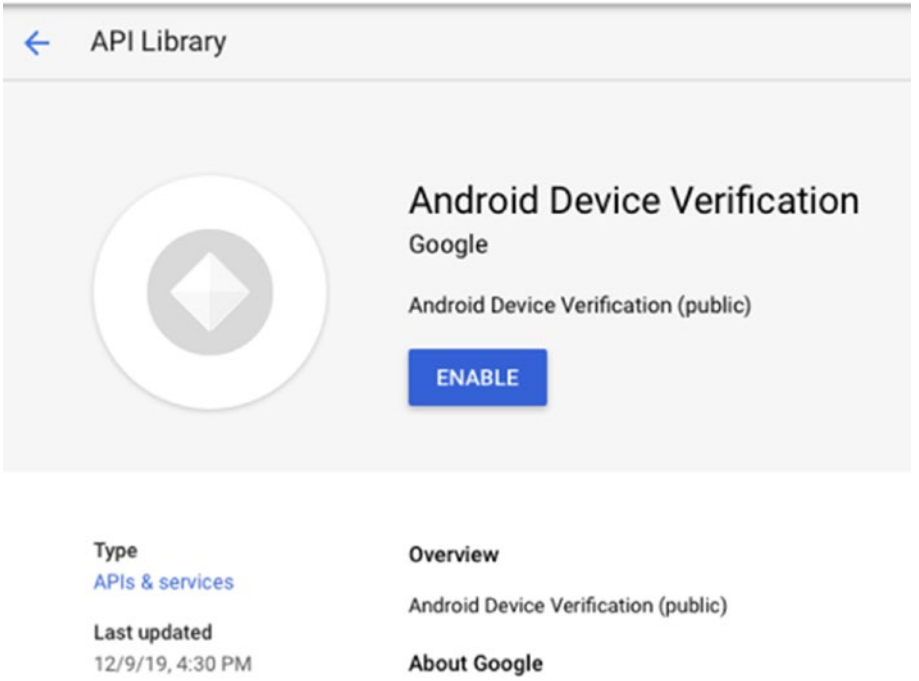


Figure 3-4. *Enabling the Android Device Verification API*

Once enabled, it is time to generate the API key. This can work in a few ways. If you see a screen similar to Figure 3-6 that says Add credentials to your project, then follow these steps further below. Now, if instead of the Add credentials screen, you saw a screen like the one in Figure 3-5, then click the Create Credentials as highlighted, or from the left sidebar, click the Credentials option. When you select those options, you will be brought to the screen as shown in Figure 3-7, and you can continue the process from there once again.

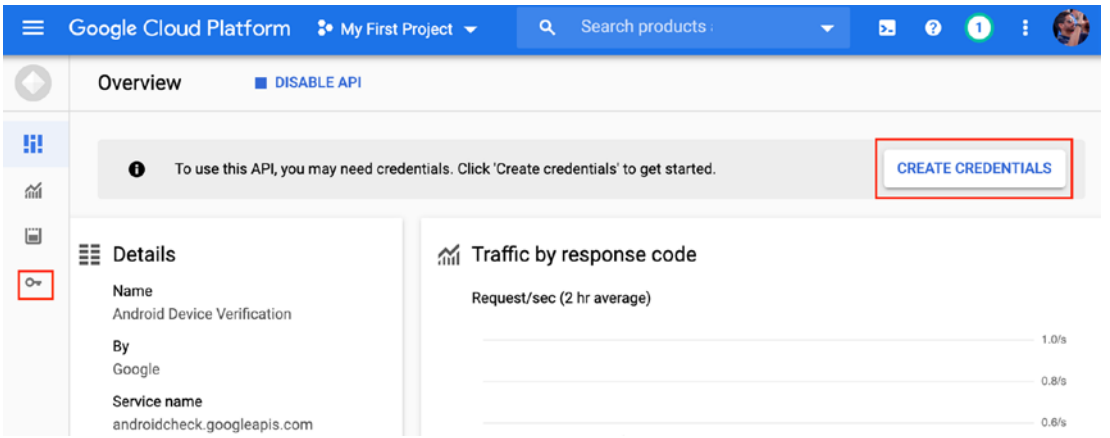


Figure 3-5. *Alternate approach to creating API keys*

In the drop-down that is entitled Which API are you using? Select *Android Device Verification* as shown and then click the *What credentials do I need?* button. An API key should automatically be generated for you as shown in Figure 3-7. Copy this key and use this in your Android project.

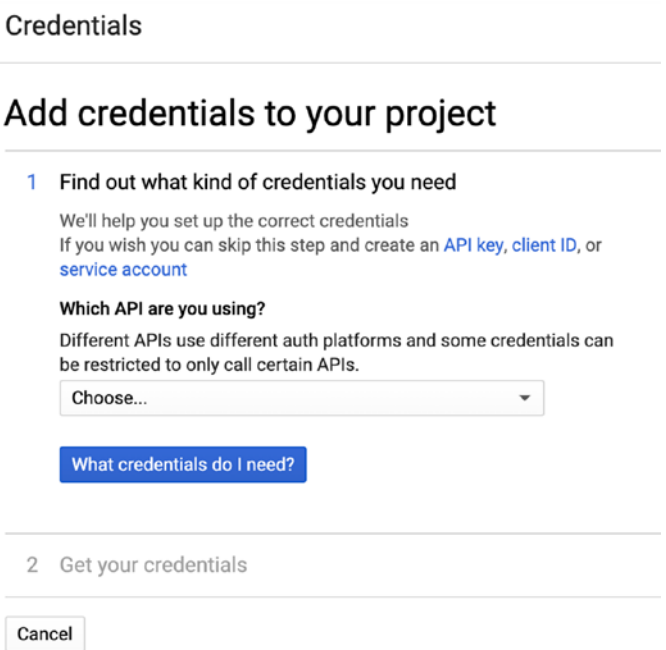


Figure 3-6. *Screen showing Add credentials to your project*

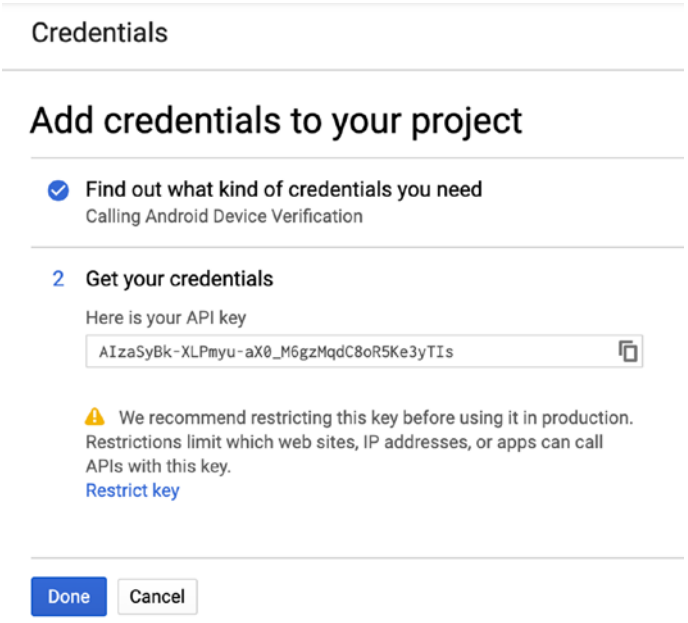


Figure 3-7. *API key being created*

Building the Back End

Once again, the back-end source code will be provided, but I want to share with you the basics. The back end can be written in multiple languages; I have written mine in Go. I will give you a pseudocode interpretation of the endpoints so that you can effectively build it in any other language if required.

Our Go back end: main.go

```
001: package main
002:
003: import (
004:     "bytes"
005:     "crypto/rand"
006:     "crypto/sha256"
007:     "crypto/x509"
008:     "encoding/base64"
```

```
009:     "encoding/binary"
010:     "encoding/json"
011:     "encoding/pem"
012:     "fmt"
013:     "io"
014:     "log"
015:     "net/http"
016:     "strings"
017:     "time"
018:
019:     jose "github.com/square/go-jose"
020: )
021:
022: type Resp struct {
023:     JWSResult string `json:"jws_result"`
024:     Action string `json:"action"`
025: }
026:
027: func buildCert(data string) string {
028:     out := new(bytes.Buffer)
029:     ctr := 0
030:     out.WriteString("-----BEGIN CERTIFICATE-----\n")
031:     for x := 0; x < len(data); x++ {
032:                 if ctr == 64 {
033:                           out.WriteRune('\n')
034:                           ctr = 0
035:                 }
036:                 out.WriteByte(data[x])
037:                 ctr++
038:     }
039:     out.WriteString("\n-----END CERTIFICATE-----")
040:     return out.String()
041: }
042:
043: func (r *Resp) verify() ([]byte, error){
044:     certs := strings.Split(r.JWSResult, ".")[0] + "=="
```

```
045:     out, err := base64.StdEncoding.DecodeString(certs)
046:     if err != nil {
047:                 return nil, err
048:     }
049:     var certObj map[string]interface{}
050:     if err := json.Unmarshal(out, &certObj); err != nil {
051:                 return nil, err
052:     }
053:     pemCert := buildCert(certObj["x5c"].([]interface{})[0].(string))
054:     block, _ := pem.Decode([]byte(pemCert))
055:     derCert, err := x509.ParseCertificate(block.Bytes)
056:     if err != nil {
057:                 return nil, err
058:     }
059:     object, err := jose.ParseSigned(r.JWSResult)
060:     if err != nil {
061:                 return nil, err
062:     }
063:     oout, err := object.Verify(derCert.PublicKey)
064:     if err != nil {
065:                 return nil, err
066:     }
067:     return oout,nil
068: }
069:
070: func (r *Resp) getDecodedResult(payload []byte) interface{} {
071:     var obj interface{}
072:     if err := json.Unmarshal(payload, &obj); err != nil {
073:                 return nil
074:     }
075:     return obj
076: }
077:
078: func generateNonce(w http.ResponseWriter, req *http.Request) {
```

```
079:     // Here we do minimal error checking and do not bother whether we
            get a GET or POST
080:
081:     rbytes := make([]byte, 8)
082:     ctime := make([]byte, 8)
083:     binary.PutUvarint(ctime, uint64(time.Now().Unix()))
084:     _, err := io.ReadFull(rand.Reader, rbytes)
085:     if err != nil {
086:                 http.Error(w, err.Error(), http.
                    StatusInternalServerError)
087:     }
088:     hash := sha256.New()
089:     hash.Write(rbytes)
090:     hash.Write(ctime)
091:     fmt.Fprintf(w, `{"nonce":"%s"}`, base64.StdEncoding.
         EncodeToString(hash.Sum(nil)))
092:
093: }
094:
095: func validate(w http.ResponseWriter, req *http.Request) {
096:     var response Resp
097:     if err := json.NewDecoder(req.Body).Decode(&response); err != nil {
098:                 http.Error(w, err.Error(),
                    http.StatusInternalServerError)
099:                 return
100:     }
101:     payload, err := response.verify()
102:     if err != nil{
103:                 fmt.Println("Verification Failed")
104:                 fmt.Fprintf(w, `{"validation":false}`)
105:     }
106:
107:     jwsResult := response.getDecodedResult(payload).(map[string]
         interface{})
```

```
108:     ctw := jwsResult["ctsProfileMatch"].(bool)
109:     bas := jwsResult["basicIntegrity"].(bool)
110:
111:     if !ctw || !bas {
112:             fmt.Println("Verification Failed")
113:             fmt.Fprintf(w, `{"validation":false}`)
114:     } else {
115:             fmt.Println("Verification Succeeded")
116:             fmt.Fprintf(w, `{"validation":true}`)
117:     }
118: }
119:
120: func main() {
121:
122:     http.HandleFunc("/nonce", generateNonce)
123:     http.HandleFunc("/validate", validate)
124:
125:     log.Fatal(http.ListenAndServeTLS(":8443", "fullchain.pem",
        "privkey.pem", nil))
126:
127: }
128:
```

A requirement for using Volley in Android on an actual device is that you are forced to use HTTPS rather than a plaintext HTTP. Therefore, you will need to get a server certificate to run the back end. I cover how to create a server certificate in Chapter 9. In our preceding code, you will notice two endpoints – `/nonce` and `/validate`. These two generate and reply with a unique nonce and validate a returned SafetyNet result, respectively.

Our nonce generation code is on lines 78–93. We take the current timestamp of the server in seconds, concatenate a set of random bytes, and then generate a SHA256 hash with that data. We then Base64 encode the SHA256 hash that we generated. We then return this Base64 string as our nonce.

Pseudocode for the Back End

The API endpoint that generates the nonce can be something like

1. Get server timestamp in seconds, convert to bytes, and concatenate 8 random bytes.
2. SHA256 hash the number from 1 and Base64 encode it before returning it to the requester.

The API endpoint to validate the JWS Result

1. Split the JWS Result into three parts with separator as the "." character.
2. Base64 decode the first part, then take the first certificate in the array of the label "x5c".
3. Extract the public key from 2 as pubkey.
4. Base64 decode the second part from 1.
5. Using pubkey and the algorithm from the label "alg" of point 1, verify the first two parts from point 1. They should match the third part from point 1.
6. On correct verification from point 1, verify that the labels `ctsProfileMatch` and `basicIntegrity` are both true. If not, the device has been tampered with.

Validation

When Android SafetyNet responds to your attestation request, you get a long Base64 encoded string. I found that there was insufficient information on the Google website that details the structure of the response. Instead, there is a link that sends you to the RFC [https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-36] for JSON Web Signature (JWS) which is essentially the result that is given to us. A JSON Web Signature (JWS) is serialized with a URL-safe encoding called JSON Compact Serialization and is represented as three separate sections concatenated to each other and separated by the “ . ” character. The first section is the Protected Header, then next is the Payload, and the third is the Signature. Another alternative representation of JWS

is the JWS JSON Serialization which has four parts. These are Protected Header, Unprotected Header, Payload, and Signature. In my experiments so far, the result has always been the JSON Compact Serialization with the three sections. I was unable to find out on Google's documentation pages the type of response returned. In any case, it may make sense to check for the number of sections there are. I would expect, at minimum, some detail outlining the format that was returned, but as of writing this book, there is none.

One of the first steps that needs to be done is to verify that the included signature can be verified. To do this, the first two blocks (the Protected Header and Payload) are hashed and signature obtained. This is done by first decoding the header and extracting the certificate that is included. The public key of this certificate is then extracted and is used to verify whether it matches the included signature. See Figure 3-8.

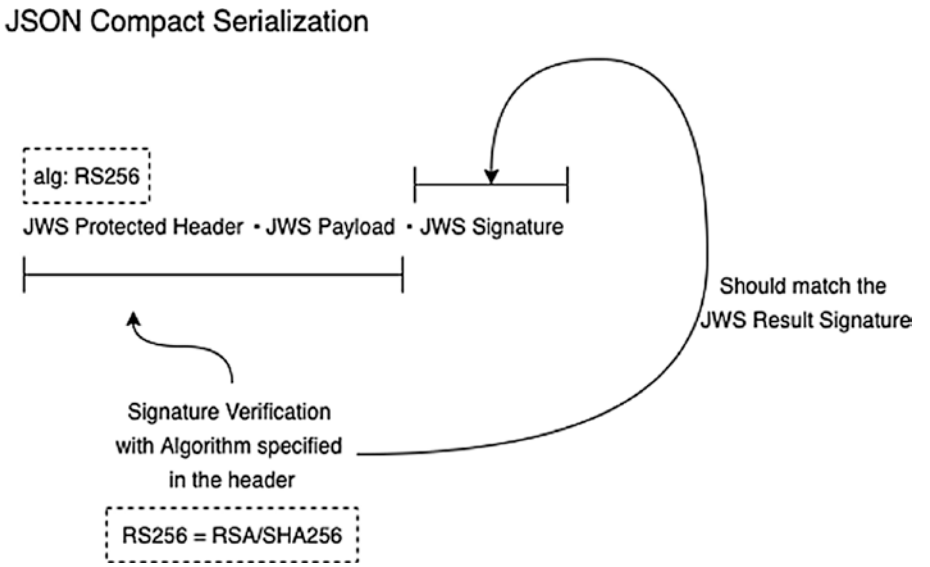


Figure 3-8. *Verifying the signature*

The JWS Protected Header when Base64 decoded contains a JSON structure similar to the one shown in Figure 3-9. Essentially, it contains two parts, the algorithm that was used to generate the signature and the collection of certificates used on the server to generate the signature. The first signature in the array is always the certificate that was used to generate the signature. Subsequent certificates can be used to verify the authenticity of the signing certificate.

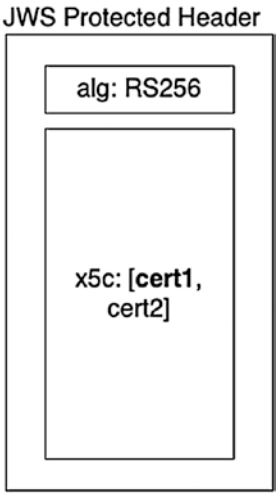


Figure 3-9. *The JWS Protected Header structure*

If the signature can be successfully verified, then you can trust the payload that was received. In our back-end code, you will see the signature verification process beginning at line 43. We use the Go library called `go-jose` developed and open sourced by Square to do our verification. If at any point during the verification process we generate an error, then we mark the entire process as failing the verification process. This is a full-on fail closed situation. You may want to consider how you want to handle the failure as you may want to be less strict. Once we verify that we can trust our payload successfully, we move on to the actual response to our attestation request from Google.

The Payload

Here is what the payload looks like when I ran the app on my Pixel 3 XL:

```
01: {
02:   "nonce": "Rk5XZkVRY1NYRm81WEJ3RGlvVUQ5TOdMcm9yRzE5NXpyQOdGMOc1VncxWTO=",
03:   "timestampMs": 1592240053813,
04:   "apkPackageName": "com.redteamlife.aas2.aas2attest",
05:   "apkDigestSha256": "rs1zB8eAT1RJXRNxhfYOtGGt4SclVYpcRiVOxMjWhoI=",
06:   "ctsProfileMatch": true,
07:   "apkCertificateDigestSha256": [
08:     "lSUzNazD8+CoMyjoOWaO5bxVuzWycKDOYq17UrnEZ3o="
09:   ],
```

```
10:   "basicIntegrity": true,
11:   "evaluationType": "BASIC,HARDWARE_BACKED"
12: }
```

Line 2 has the original nonce that we generated at the beginning of this attestation process. In our example, we are not doing anything with our nonces, but they ideally should be generated, placed in a temporary database such as Redis or a file, and then marked as used when the request completes its full cycle. The purpose of the nonce is so that an attacker can't send the same request to us to find out more information about how our payloads are structured. If we detect the same nonce being sent to us after we marked it as used, then we know not to take further action and to drop the request.

Tip You can further enhance the effectiveness of the nonce by giving it a certain time to live. So the instant that we generate a nonce, we also generate an expiry time of, let's say, two minutes. This means that we have a two-minute window in which to receive our attestation response. Anything arriving later than that is disregarded, and the client has to start a new attestation cycle once again.

We are mostly interested in lines 6 and 10 – the `ctsProfileMatch` and `basicIntegrity` responses. These two parameters determine the integrity of the device that our app is running on. A false on one or both of those should put you on alert because this indicates that the device is tampered with. Take a look at Figure 3-10 which shows a table that gives reasons why one or more of the parameters are set to true or false.

Device Status	Value of ctsProfileMatch	Value of basicIntegrity
Certified, genuine device that passes CTS	`true`	`true`
Certified device with unlocked bootloader	`false`	`true`
Genuine but uncertified device, such as when the manufacturer doesn't apply for certification	`false`	`true`
Device with custom ROM (not rooted)	`false`	`true`
Emulator	`false`	`false`
No device (such as a protocol emulating script)	`false`	`false`
Signs of system integrity compromise, one of which may be rooting	`false`	`false`
Signs of other active attacks, such as API hooking	`false`	`false`

Figure 3-10. *Integrity states and their explanation. Taken from https://developer.android.com/training/safetynet/attestation*

If you look at our back-end code on lines 111–117, you will see where we decide whether to validate the request or not. We are using a strict approach here and only returning true if a device reports that both ctsProfileMatch and basicIntegrity are true. Further information regarding the parameters and their explanations can be found here: https://developer.android.com/training/safetynet/attestation#use-response-server.

Can This Be Bypassed?

This is indeed a good question, and until around April 2020 or so, the answer would have been a resounding yes (Figure 3-11). The rooting framework called Magisk (we dive deeper in Chapter 8) has an option that can hide itself and the fact that the device is rooted from Android SafetyNet. This has now changed thanks to how Google does the data gathering regarding the device state. As yet, I do not have the full picture, but have read that it has something to do with the use of the Trusted Execution Environment (TEE) which essentially hides key information from the Android operating system.

Figure 3-11. *The author of Magisk, John Wu, tweeting that it is no longer possible to bypass SafetyNet*

So, Why Don't Many People Use SafetyNet?

I think because it's quite cumbersome and gives the developer yet another set of code to maintain outside of the normal set of app features. Also, I think for the longest time, it was possible to report false information about the device itself to SafetyNet if you used frameworks like Magisk. So, people most likely decided to skip the extra work that they would have had to do in getting SafetyNet up and running. I think differently, however, especially now with the more effective way of preventing manipulation even by frameworks such as Magisk. I think having SafetyNet Attestation as a complement to your app's functionality is certainly a good thing and gives you a greater degree of control when it comes to protecting your app.

CHAPTER 4

Securing Your Apps at Scale

I wanted to cover some techniques that I have seen and used when protecting Android apps. I titled the chapter with the words "at scale" because when I was at Gojek, everything was at scale – the users, the engineering teams, the threats. As I may or may not have mentioned in my previous book, a healthy dose of paranoia will serve you well. Personally, I don't think that you as a developer stand a chance against the reverse engineers. They are always ahead, and they are always at an advantage. I think it is somewhat of a waste of time spending a disproportionate amount of time trying to secure your apps. Then all you have built is an app with 20% actual features and 80% security. No balance. It's like the engineering team that focuses so much on testing that you're devoting all your time and energy to tests and not to the code itself. I recall Zed Shaw (the creator of the Mongrel web server for Ruby web applications) once wrote about this in a scathing piece leveled at the Ruby on Rails community entitled "Rails is A Ghetto." He was mentioning a well-known software development company at the time, and he said:

> *During their operations they seem to focus entirely on the process, but very little on the quality of the code. Sorry guys, but having a 1:4 code:test ratio is not focusing on code quality. It's focusing on test quality.*
>
> —Zed A. Shaw

What's my point? Well, it's simple. Come to terms with the fact that your app will be hacked. Then take steps to secure your app to first protect your user's data from getting into the wrong hands. Then also focus on financial loss that your business will incur through a flaw in your app and secure that. Lastly, take steps to protect your intellectual property. Here, I would like to pause a little and talk about the intellectual property

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_4

aspect. In present times, it is highly unlikely that you will have some magical intellectual property that you are dying so hard to protect. If you did, you should apply for a patent. I think that apps these days offer so little in the way of sheer uniqueness and that it is very easy to look at how one app works and instantly replicate what it does. If you have a proprietary bit of code that makes performance improvements tenfold, then for sure protect that, but let's face it, you're not going to have code like that. Your app should always simply be a snazzy UI that talks to the powerful back-end servers. Hide all your secrets in your back-end code; don't put them up front in your mobile app. So, what do I mean when I say protect your awesome performance improvement engine? I'm talking about obfuscation. Code obfuscation has its place. To me, code obfuscation keeps out the "drive-by hackers" – the folks that are bored and just want to mess with your app and see what they can do to it. Determined, purposeful hackers, however, will stop at nothing to break your obfuscation. And they will break it because they have full control of where their app runs. Essentially, you're sending your APK into an extremely hostile environment. No longer are the days where your infrastructure protection wall will keep your systems behind it nice and cozy. These days, you're sending your app out into enemy territory. If you keep that thought in your head when you build and deploy your apps, you should automatically find yourself changing your development and deployment techniques as well.

But let's not get ahead of ourselves. In this chapter, I will pick out the various mechanisms of protecting your app and then take a deeper look at each of them by showing you what each of the apps looks like under different conditions like decompiling and debugging. We will then also have a discussion around what a hacker can do further to break the app and what a developer can do further to protect it.

Static Source Code Security Analysis

Let's start at the beginning – the place where source code is born, your developer team. We should all trust our developers, right? I'll take a hard pass at that and go back to the tried and tested "trust but verify" ethos. We all want to believe that our developers are able to churn out line after line of secure code, but this is not the case. This isn't a bad thing. I write really bad, insecure code myself. Therefore, as a security practitioner, we should strive to see past the stigma that the developer has received, which is that they can't write secure code. Or, to put it more accurately, the unicorn developer that writes good code is talked about but is never found in our own teams. Let's turn our attention

back to my bad code. The important point about this is that I am very aware that I write bad code. While this may stem from my issues with low esteem, I feel that having this mindset has forced me to double- and triple-check my work. One of the ways I do it is by having it security tested by an external party. Whether this is my own security red team or whether it is an outside vendor, it is important that an independent, objective security practitioner look at my work. This is still done after the fact of launching my bad code though. An interim step that I should probably take (the missing one in Figure 4-1), and one that I recommend you seriously consider if you have a large development team, is to perform some static application security testing or SAST. Now this is just a fancy way of saying check the security of the static parts of your app like source code, design choices, or binaries. I use SAST in the context of checking your source code for common vulnerabilities or instances of bad bits of code that can lead to vulnerabilities.

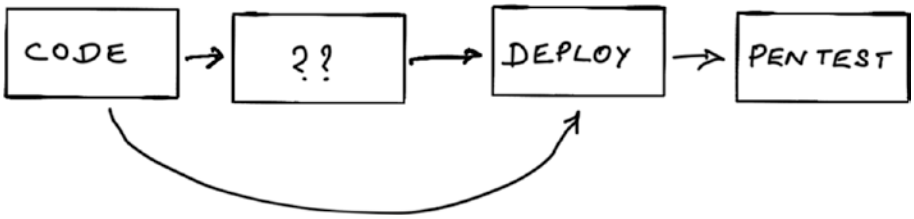


Figure 4-1. *My missing source code review diagram*

The idea of having a source code review process is to address the low-hanging fruit – especially the mistakes that developers make repeatedly because it has become part of their way of working. For example, if you take Java on Android, some key things you may want to look out for would be things like

- Using `addJavascriptInterface()` which allows JavaScript code to access the main application
- SQL injection
- Cross-site scripting (XSS) in a WebView

You could and should even extend this process to your back-end code as well. Theoretically, you could do all this with a separate team of developers hired only to review code, but I'm talking here about using automated tools to get the job done. There are many tools that help you get this part done. You will have to choose the correct one weighing effectiveness, ease of setup, and cost. The OWASP website has a list of both free and commercial products that can help with this [https://owasp.org/www-community/Source_Code_Analysis_Tools].

"How would I implement this?" would be a possible question that you may have at this point. It is my firm opinion that it is better to have the automated source code review cycle done in a transparent manner. For me, the most ideal would be to place this step as the final part of your build process in your continuous integration cycle. This step can come before you do your final testing prior to building your software (you ARE writing tests for your code right?). The workflow should be designed so that if a designated category of security bugs is found then the entire build process will fail. It can only work if all security flaws in the code are rectified. A conceptual diagram in Figure 4-2 shows this.

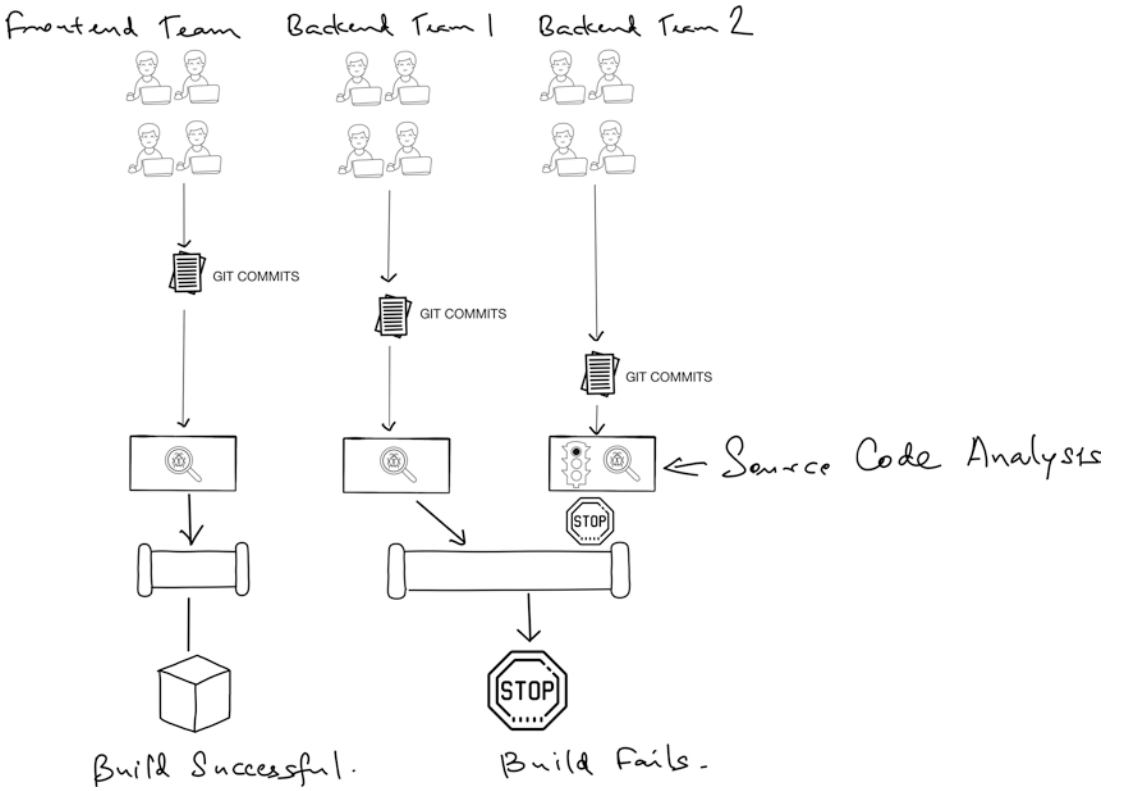


Figure 4-2. *Setting up a source code analysis on the build pipeline*

Third-Party Libraries or Dependencies

Another aspect to consider is the security of your third-party dependencies. It makes little sense to have to write your applications all from scratch, and chances are that someone has already written the functionality you want and open sourced it. It makes

sense to see how secure a third-party dependency is if you can get access to the source code. While open source projects are great, if you are running a very critical system handling thousands of users' personal, private, or financial data, then you will seriously also want to consider the dependencies you are using and include them in your testing as well where possible.

Developer Training

So far, the solutions presented do not include training. This may often be overlooked but is a key point to helping your developers churn out secure code. I advise information security training which gives your developers an insight into what hostile environments and attacks their finished product is subject to. This should give them an understanding of what areas of weakness in their code are being exploited. I also firmly believe that openly discussing the failures without assigning blame is extremely valuable to facilitate learning and retention. Having something similar to a retrospective could be beneficial. This is where all the developers of a project are gathered for a workshop after a release or security assessment is performed. The team goes through all the mistakes made and analyzes how the vulnerability came to be created. While time-consuming, I think these are important processes that should not be neglected in favor of the chase for new features or growth.

Obfuscation

A very popular way of protecting Android apps from being abused, for lack of a better term, is to obfuscate it. You may have heard the term mentioned in discussions and arguments on whether to obfuscate or not. Simply put, obfuscation is a process used to purposely render source code unreadable to humans – the premise being that if you decompile an application that has been obfuscated, the resulting source code would be confusing and unreadable to someone looking at it. The key here is that it is unreadable to the human, but a machine can understand and process it. As a quick example, there is an obfuscation called class renaming. In class renaming, the obfuscation involves changing the name of a Java class from, let's say, `com.redteamlife.LicenseGenerator` to `a.a.b`. The code will still work because the virtual machine it runs on understands it and doesn't necessarily care about what you name your classes. As developers, we all strive to write code in a manner that is easily readable by ourselves or our team

members. Thus, we pick names of variables and functions that are more recognizable to the human. In some instances, like our preceding example, it also shrinks the code which means your final app is much smaller. One key point to keep in mind is that obfuscation will make it difficult for reverse engineers to understand your code, but it will not make it impossible to reverse engineer the code. Eventually, given enough time, a reverse engineer can make complete sense of your obfuscated code. Let's look at some techniques of obfuscation.

String Encryption

As the name implies, string encryption is when human-readable strings are encrypted with some sort of secret key. Thus, when the software is decompiled, at first glance, the reverse engineer or attacker will see some garbled strings which won't make sense. For example:

```
val license = "abcd-1234-aasd"
```

could appear as

```
val license = DecryptString("RCGgU9tvvbaoXVpU")
```

The `DecryptString` method will be added to your APK by the obfuscator and will execute each time your app runs. If you're wondering, "Hey won't this impact my performance?", you would be right. All obfuscators add a layer of performance overhead to your app. This is why you will find most of the commercial obfuscation vendors telling you to only obfuscate the classes or parts of your code that are sensitive. There's no need to obfuscate your whole app as this could add time to its final execution and then impact user experience.

Class Renaming

As we saw in our earlier example, class renaming is when the name of the class itself is renamed. This can extend to field renaming and method renaming also for maximum effectiveness. A non-obfuscated APK is shown in Figure 4-3, and what that looks like after it is run through an obfuscation routine is shown in Figure 4-4. Class renaming could also be referred to as minification and can fall under the performance improvement category as well because it effectively shrinks the size of your APK.

Figure 4-3. *A normal class hierarchy*

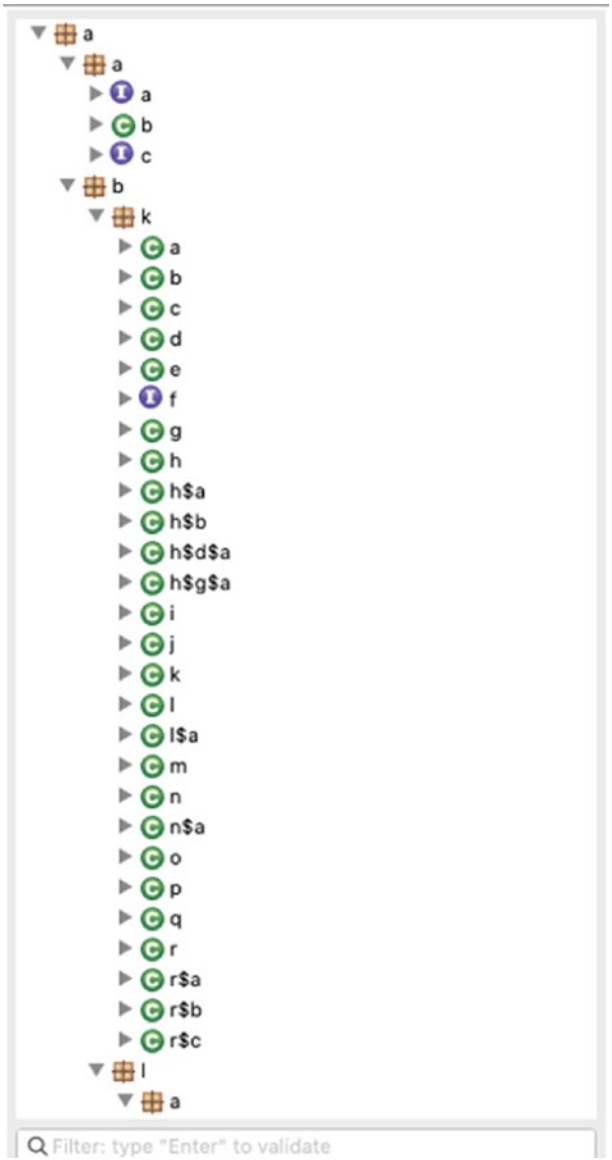


Figure 4-4. *The same class hierarchy in Figure 4-3 but after class renaming/minification*

Spaghetti Code/Control Flow Alteration

We all know as developers that writing spaghetti code is bad, unreadable, and fraught with triggering errors in the program execution. Here, in obfuscation land, we purposely write this bad spaghetti code. The reason? To send a reverse engineer around in circles literally. Again, this has to be done with care because it could adversely affect the app's performance.

NOP and Code Injection

A NOP or No Operation is an instruction of machine code where nothing happens. The processor will just move on to the next line if it encounters a NOP in a program. By liberally spraying NOPs into the bytecode, it is possible to create a much longer dead listing of disassembled code. So, it would take a lot of scrolling if you were to reverse engineer a NOP-ridden bit of code. Of course, if you have a good disassembler, you can merely double-click a piece of code and quickly jump to it rather than painstakingly scrolling. Still, NOPs make it annoying to look at and understand reverse engineered code.

Another injection point other than NOPs can even be dummy code injection. So liberally spraying your APK with dummy bits of code can keep a reverse engineer busy for a long time. He may spend an inordinately long time understanding a method only to realize it has absolutely nothing to do with the actual code he wants to examine.

Which Obfuscator to Use

I am not going to recommend any obfuscators to use here because I think you will have to weigh in the features vs. cost. I can tell you that I have used both DexGuard and Arxan, and both packages are not cheap. Generally, effective obfuscation involves consistently pushing boundaries and coming up with new and innovative mechanisms on how to obfuscate code. This endeavor is not cheap and thus requires a lot of research. Therefore, I understand when companies with highly effective obfuscation platforms charge top dollar. With Android, there is a packaged obfuscator called ProGuard that is supplied. ProGuard in this case acts more like an optimizer instead of an obfuscator as it doesn't even have string encryption. For this, you will have to buy its fully featured sibling DexGuard. In the following examples, I will show you what a ProGuarded program looks like after decompiling. I will also show you what a program looks like after we apply a few tweaks using a free obfuscator called Obfuscapk [www.sciencedirect.com/science/article/pii/S2352711019302791].

Our Base Program

Let's first write a very basic program in Kotlin language with which to test out our obfuscation. In Figure 4-5, you can see what the output of the app looks like. Here's the set of classes:

MainActivity.kt source code

```
01: package com.redteamlife.aas2.aas2obfuscate
02:
03: import androidx.appcompat.app.AppCompatActivity
04: import android.os.Bundle
05: import android.widget.TextView
06: import java.util.*
07:
08: class MainActivity : AppCompatActivity() {
09:
10:     override fun onCreate(savedInstanceState: Bundle?) {
11:         super.onCreate(savedInstanceState)
12:         setContentView(R.layout.activity_main)
13:
14:         val testModule = TestModule()
15:
16:
17:         if (testModule.isOddDay()) {
18:             findViewById<TextView>(R.id.oddDay).text = "It's an odd day
                today."
19:         }
20:
21:         val nuts = testModule.getNuts()
22:         val loop : TextView = findViewById(R.id.loop)
23:         for (type in nuts){
24:             loop.append(type+"\n")
```

```
25:         }
26:     }
27: }
28:
```

TestModule.kt source code

```
01: package com.redteamlife.aas2.aas2obfuscate
02:
03: import android.util.Log
04: import java.util.*
05:
06: class TestModule{
07:     fun logSomething(){
08:         Log.d("aas2obfuscate","This is something")
09:     }
10:
11:     fun isOddDay()  : Boolean {
12:         val calendar : Calendar = Calendar.getInstance()
13:         return calendar.get(Calendar.DAY_OF_MONTH) % 2 != 0
14:     }
15:
16:     fun getNuts(): Array<String> {
17:         return arrayOf("almonds","peanuts","cashews","hazelnuts")
18:     }
19: }
```

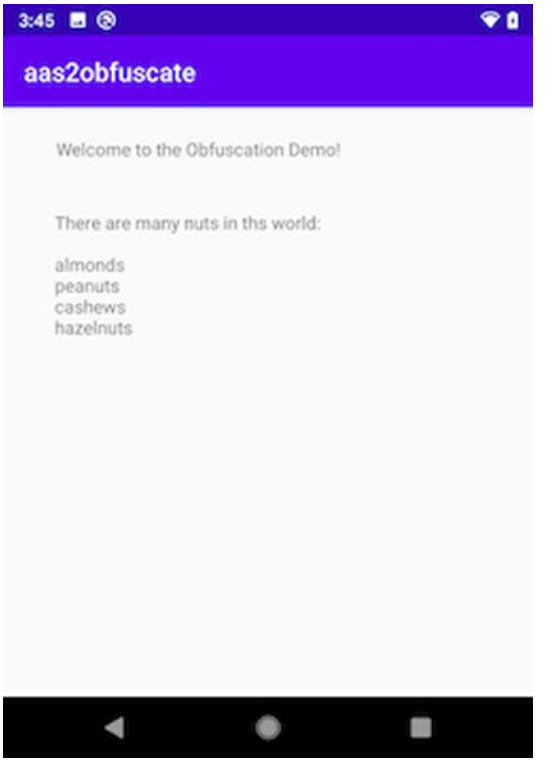


Figure 4-5. *Output from our sample program*

As you can see, our example program is a little silly, contrived example to demonstrate what ProGuard does. If you look at your Android Studio's `build.gradle` file for your module, you will see code that may look like this:

```
1: buildTypes {
2:         release {
3:             minifyEnabled true
4:             proguardFiles getDefaultProguardFile('proguard-android-
               optimize.txt'), 'proguard-rules.pro'
5:
6:         }
7:     }
```

We will purposely comment out lines 3 and 4 so that when we build our final release APK, it will have no optimizations or ProGuard-related alterations. Go ahead and build that, and when it's done, call it something like `app-release-plain.apk`. Then,

uncomment the two lines and build it again so that you get the version with ProGuard enabled. Call this one `app-release-proguard.apk`.

First, let's take a look at what we spoke about in terms of shrinking size. If you do a directory listing on your files, you will see something like this:

```
1: ➜  ls -al
2: total 6016
3: drwxr-xr-x   6 sheran  staff      192 May 21 16:00 .
4: drwxr-xr-x  21 sheran  staff      672 May 20 22:39 ..
5: -rw-r--r--   1 sheran  staff  2039093 May 21 16:00 app-release-plain.apk
6: -rw-r--r--   1 sheran  staff  1029194 May 21 15:59 app-release-proguard.
   apk
7: ➜
```

You can clearly see that our ProGuard APK is half the size of our plain nonoptimized APK. So right off the bat, you can see the benefits of running your APK through ProGuard. For this time's comparison, let's use some free tools. You will need to have dex2jar [https://github.com/pxb1988/dex2jar/releases/tag/2.0] and JD-GUI[https://github.com/java-decompiler/jd-gui/releases/tag/v1.6.6] for this next step. To set up dex2jar, you just have to unzip the file that you download into its own directory. For JD-GUI, download the version that suits your operating system and either unzip or untar the file. For OS X, I have to untar the file as follows:

```
tar xf jd-gui-osx-1.6.6.tar
```

This will create a `jd-gui-osx-1.6.6` directory in the current directory with an OS X app that you can just double-click from Finder.

First, let's convert our plain app into a jar. From your dex2jar directory, run

```
1: ➜  ./d2j-dex2jar.sh ~/Documents/android_research/aasobfuscate/app-
   release-plain.apk -o ~/Documents/android_research/aasobfuscate/app-
   release-plain-dex2jar.jar
2: dex2jar /Users/sheran/Documents/android_research/aasobfuscate/
   app-release-plain.apk -> /Users/sheran/Documents/android_research/
   aasobfuscate/app-release-plain-dex2jar.jar
3: ➜
```

This will convert the classes.dex file within the APK and write a jar file to the directory where you originally saved your APK. If you skip the -o option, then the file is output to the current directory. Now let's fire up JD-GUI and open the jar file that we just created. Here is what the reversed code looks like:

Reversed MainActivity.class file

```
01: package com.redteamlife.aas2.aas2obfuscate;
02:
03: import android.os.Bundle;
04: import android.view.View;
05: import android.widget.TextView;
06: import androidx.appcompat.app.AppCompatActivity;
07: import java.util.HashMap;
08: import kotlin.Metadata;
09: import kotlin.jvm.internal.Intrinsics;
10:
11: @Metadata(bv = {1, 0, 3}, d1 = {"\000\030\n\002\030\002\n\002\030\002\
    n\002\b\002\n\002\020\002\n\000\n\002\030\002\n\000\030\0002\0020\001B\
    005\006\002\020\002J\022\020\003\032\0020\0042\b\020\005\032\004\030\
    0010\006H\024\006\007"}, d2 = {"Lcom/redteamlife/aas2/aas2obfuscate/
    MainActivity;", "Landroidx/appcompat/app/AppCompatActivity;", "()V",
    "onCreate", "", "savedInstanceState", "Landroid/os/Bundle;", "app_
    release"}, k = 1, mv = {1, 1, 16})
12: public final class MainActivity extends AppCompatActivity {
13:   private HashMap _$_findViewCache;
14:
15:   public void _$_clearFindViewByIdCache() {
16:     HashMap hashMap = this._$_findViewCache;
17:     if (hashMap != null)
18:       hashMap.clear();
19:   }
20:
21:   public View _$_findCachedViewById(int paramInt) {
22:     if (this._$_findViewCache == null)
23:       this._$_findViewCache = new HashMap<>();
```

```
24:     View view1 = (View)this._$_findViewCache.get(Integer.
        valueOf(paramInt));
25:     View view2 = view1;
26:     if (view1 == null) {
27:       view2 = findViewById(paramInt);
28:       this._$_findViewCache.put(Integer.valueOf(paramInt), view2);
29:     }
30:     return view2;
31:   }
32:
33:   protected void onCreate(Bundle paramBundle) {
34:     super.onCreate(paramBundle);
35:     setContentView(2131361820);
36:     TestModule testModule = new TestModule();
37:     if (testModule.isOddDay()) {
38:       View view1 = findViewById(2131165299);
39:       Intrinsics.checkExpressionValueIsNotNull(view1,
          "findViewById<TextView>(R.id.oddDay)");
40:       ((TextView)view1).setText("It's an odd day today.");
41:     }
42:     String[] arrayOfString = testModule.getNuts();
43:     View view = findViewById(2131165289);
44:     Intrinsics.checkExpressionValueIsNotNull(view, "findViewById
        (R.id.loop)");
45:     TextView textView = (TextView)view;
46:     int i = arrayOfString.length;
47:     for (byte b = 0; b < i; b++) {
48:       String str = arrayOfString[b];
49:       StringBuilder stringBuilder = new StringBuilder();
50:       stringBuilder.append(str);
51:       stringBuilder.append("\n");
52:       textView.append(stringBuilder.toString());
53:     }
54:   }
55: }
56:
```

Reversed TestModule.class file

```
01: package com.redteamlife.aas2.aas2obfuscate;
02:
03: import android.util.Log;
04: import java.util.Calendar;
05: import kotlin.Metadata;
06: import kotlin.jvm.internal.Intrinsics;
07:
08: @Metadata(bv = {1, 0, 3}, d1 = {"\000$\n\002\030\002\n\002\020\000\
    n\002\b\002\n\002\020\021\n\002\020\016\n\002\b\002\n\002\020\013\
    n\000\n\002\020\002\n\000\030\0002\0020\001B\005\006\002\020\0
    02J\021\020\003\032\b\022\004\022\0020\0050\004\006\002\020\00
    6J\006\020\007\032\0020\bJ\006\020\t\032\0020\n\006\013"}, d2 = {"Lcom/
    redteamlife/aas2/aas2obfuscate/TestModule;", "", "()V", "getNuts", "",
    "", "()[Ljava/lang/String;", "isOddDay", "", "logSomething", "",
    "app_release"}, k = 1, mv = {1, 1, 16})
09: public final class TestModule {
10:   public final String[] getNuts() {
11:     return new String[] { "almonds", "peanuts", "cashews", "hazelnuts"
};
12:   }
13:
14:   public final boolean isOddDay() {
15:     boolean bool;
16:     Calendar calendar = Calendar.getInstance();
17:     Intrinsics.checkExpressionValueIsNotNull(calendar, "Calendar.
        getInstance()");
18:     if (calendar.get(5) % 2 != 0) {
19:       bool = true;
20:     } else {
21:       bool = false;
22:     }
23:     return bool;
24:   }
25:
```

```
26:   public final void logSomething() {
27:     Log.d("aas2obfuscate", "This is something");
28:   }
29: }
30:
```

Apart from a few extra methods in the `MainActivity.class` file, it looks pretty similar to what we had. Do note that the reverse engineering or decompiling in this method will convert your code into Java code. So that's why some of it will look different from Kotlin. Now, let's do the same for the ProGuard file. Interestingly, you won't find your `TestModule.class` file when ProGuard is used. It will be combined into your `MainActivity.class` file. Here's the reversed code:

Reversed MainActivity.class file using ProGuard

```
01: package com.redteamlife.aas2.aas2obfuscate;
02:
03: import a.b.k.e;
04: import android.os.Bundle;
05: import android.view.View;
06: import android.widget.TextView;
07: import c.a.a.a;
08: import java.util.Calendar;
09:
10: public final class MainActivity extends e {
11:   public void onCreate(Bundle paramBundle) {
12:     super.onCreate(paramBundle);
13:     setContentView(2131361820);
14:     Calendar calendar = Calendar.getInstance();
15:     a.a(calendar, "Calendar.getInstance()");
16:     int i = calendar.get(5);
17:     boolean bool = false;
18:     if (i % 2 != 0) {
19:       i = 1;
20:     } else {
21:       i = 0;
22:     }
```

```
23:     if (i != 0) {
24:       View view1 = findViewById(2131165299);
25:       a.a(view1, "findViewById<TextView>(R.id.oddDay)");
26:       ((TextView)view1).setText("It's an odd day today.");
27:     }
28:     String[] arrayOfString = new String[4];
29:     arrayOfString[0] = "almonds";
30:     arrayOfString[1] = "peanuts";
31:     arrayOfString[2] = "cashews";
32:     arrayOfString[3] = "hazelnuts";
33:     View view = findViewById(2131165289);
34:     a.a(view, "findViewById(R.id.loop)");
35:     TextView textView = (TextView)view;
36:     int j = arrayOfString.length;
37:     for (i = bool; i < j; i++) {
38:       String str = arrayOfString[i];
39:       StringBuilder stringBuilder = new StringBuilder();
40:       stringBuilder.append(str);
41:       stringBuilder.append("\n");
42:       textView.append(stringBuilder.toString());
43:     }
44:   }
45: }
46:
```

Also, if you look at lines 3 and 7 and also within the code, you will see that classes have been renamed or minified to single character classes. At first glance, you won't have an idea what they do. You will have to dig through the code of each one to figure out what it is.

Now, let's take a look at what string encryption will look like if we applied it to our APK. Let's get Obfuscapk setup first. The author of Obfuscapk has conveniently set up the source and prerequisites within a Docker container. This is the easiest way to get started using it. You will need to have Docker Desktop downloaded and installed from here first: www.docker.com/get-started. Once Docker Desktop is up and running, you can go through some of their tutorials that will help you get familiar with how it works [www.docker.com/101-tutorial]. Once you have a basic understanding of how Docker works, we can go ahead and download the Obfuscapk container image:

```
01: ➜  docker pull claudiugeorgiu/obfuscapk
02: Using default tag: latest
03: latest: Pulling from claudiugeorgiu/obfuscapk
04: 6d28e14ab8c8: Pull complete
05: f87645a1e5dc: Pull complete
06: 6a33999da14c: Pull complete
07: a66020c98364: Pull complete
08: c47035224950: Pull complete
09: 804c66631fe2: Pull complete
10: f822a6acb3bf: Pull complete
11: af91560da3d3: Pull complete
12: e48f4dfefd50: Pull complete
13: Digest: sha256:8c6f3de137c08c1e6c5b1f8d632416dcc7b3848928c17d1676ccaee5
    a53c6216
14: Status: Downloaded newer image for claudiugeorgiu/obfuscapk:latest
15: docker.io/claudiugeorgiu/obfuscapk:latest
16: ➜  docker tag claudiugeorgiu/obfuscapk obfuscapk
17: ➜
```

In the preceding listing, I issue the command to pull the image from Docker Hub in line 1. After the download completes, in line 16, I give the command to shorten the name of how we refer to the container. Now, we can test if everything works by issuing this command:

```
➜  docker run --rm -it obfuscapk --help
usage: python3 -m obfuscapk.cli [-h] -o OBFUSCATOR [-w DIR] [-d OUT_APK] [-i]
                                [-p] [-k VT_API_KEY]
                                <APK_FILE>

Obfuscate an application (.apk) without needing its source code.

positional arguments:
  <APK_FILE>            The path to the application (.apk) to obfuscate

optional arguments:
  -h, --help            show this help message and exit
  -o OBFUSCATOR, --obfuscator OBFUSCATOR
```

```
                    The name of the obfuscator to use. Can be specified
                    multiple times to use more obfuscators (in sequence).
                    Allowed values are: AdvancedReflection,
...
...
...
```

You should see an output similar to the preceding one after you run the command. Since Docker runs within its own confined space, we need to be able to give it files to obfuscate. To do this, we have to mount a working directory on our local filesystem that the Docker container can access. This means supplying an extra `-v` flag (short for volume) to the Docker command and telling it to mount and map a local directory to a directory in the container. So, if I wanted to map my home directory in OS X to `"/workdir"` on the container, I would do `-v "/Users/sheran":"/workdir"`. Let's add string encryption now. We need to know which directory we last built our APK sample apps in. In my case, it is `/Users/sheran/Documents/android_research/aasobfuscate/` so I will use that. Here is the command I used:

```
➜  docker run --rm -it -v "/Users/sheran/Documents/android_research/
aasobfuscate":"/workdir" obfuscapk -p -o ConstStringEncryption -o Rebuild
-o NewSignature -o NewAlignment -d /workdir/out.apk /workdir/app-release-
plain.apk
Encrypting constant strings: 100%|████████████████████████████████
████████████████████████████████|[00:24<00:00, 98.97file/s]
Running obfuscators (NewAlignment): 100%|████████████████████████
████████████████|[02:14<00:00, 33.68s/obfuscator]
➜
```

Let's break down the command that we ran:

- docker run will run our container.
- `--rm` will clean up the filesystem of any temporary or working files that were created when we ran the command. This isn't necessarily mandatory, and you can ignore it if you want to.
- `-i` keeps the standard input `STDIN` open even if the container isn't attached so it can receive our input.

- `-t` allocates a pseudo-tty (terminal).
- `-v` is the command we spoke about earlier, which mount our local directory to the container's directory. The format is `localdir:containerdir`.
- obfuscapk is the name of the container that we tagged in the previous Docker command.
- `-p` is an Obfuscapk flag now. -p tells Obfuscapk to print out progress.
- `-o` tells Obfuscapk which obfuscator to use. You can use -o multiple times to identify the different obfuscators to use. Obfuscapk has several obfuscation mechanisms and [https://github.com/ElsevierSoftwareX/SOFTX_2019_275#-obfuscators] we are using the ConstStringEncryption obfuscator to encrypt all strings in the code. The other three obfuscators are always necessary when running Obfuscapk. They tell Obfuscapk to rebuild the APK, apply a NewSignature to it, and realign it (NewAlignment).
- `-d` specifies the destination and name of the output file you want.
- Last, we supply the name of the input file. You will note how both input and output files are referenced by the container's directory `"/workdir"` instead of the actual local name.

If you check the directory where you had your app-release-plain.apk, you should now also see an out.apk there. Let's take a look at what that looks like in JD-GUI. So same as before, we first run dex2jar:

```
➜  ./d2j-dex2jar.sh -o /Users/sheran/Documents/android_research/
   aasobfuscate/out-dex2jar.jar /Users/sheran/Documents/android_research/
   aasobfuscate/out.apk
   dex2jar /Users/sheran/Documents/android_research/aasobfuscate/out.apk ->
   /Users/sheran/Documents/android_research/aasobfuscate/out-dex2jar.jar
➜
```

Now take the resulting jar file and open it in JD-GUI. Then, navigate over to the `MainActivity.class` file in the `com.redteamlife.aas2.aas2obfuscate` package, and you should see something similar to Figure 4-6.

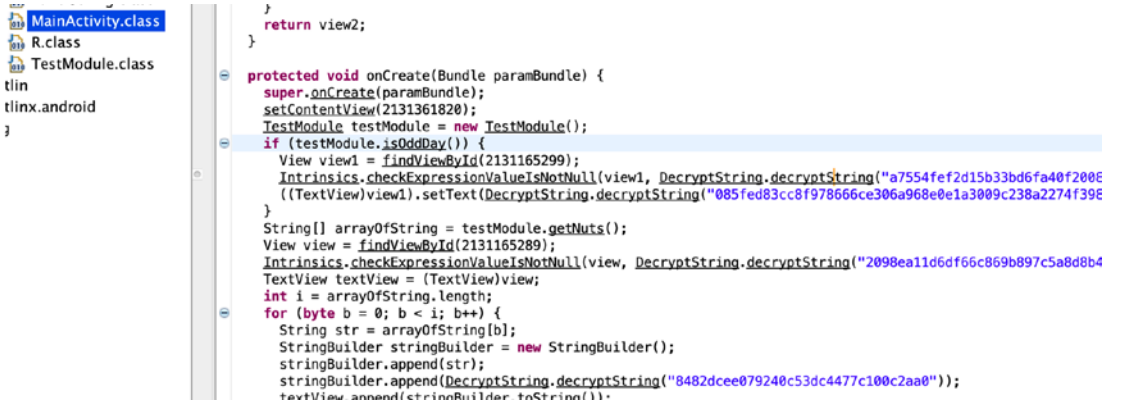


Figure 4-6. *Reversed source of MainActivity.class showing strings encrypted*

You will notice that the strings in both `MainActivity.class` and `TestModule.class` are now encrypted. You will also notice a new package called `decryptstringmanager` with a file called `Decrypt.class`. This file has been added by Obfuscapk and will be responsible for decrypting strings on the fly when the APK runs.

Caution I have yet to get the Obfuscapk string encrypted output APK running on my device. It ends up crashing which is something that can happen when you obfuscate your APKs. Sometimes changes in classes due to obfuscation can cause applications to crash.

The fact that the decrypter is visible in the code means we can also see not only how it works but also the secret key that it uses to encrypt and decrypt the strings. Some commercial decompilers have routines built in to be able to decrypt these strings on the fly as shown in Figure 4-7. Alternatively, it would not take a lot of effort to code up a decrypter that goes through the entire source tree and decrypts all the strings.

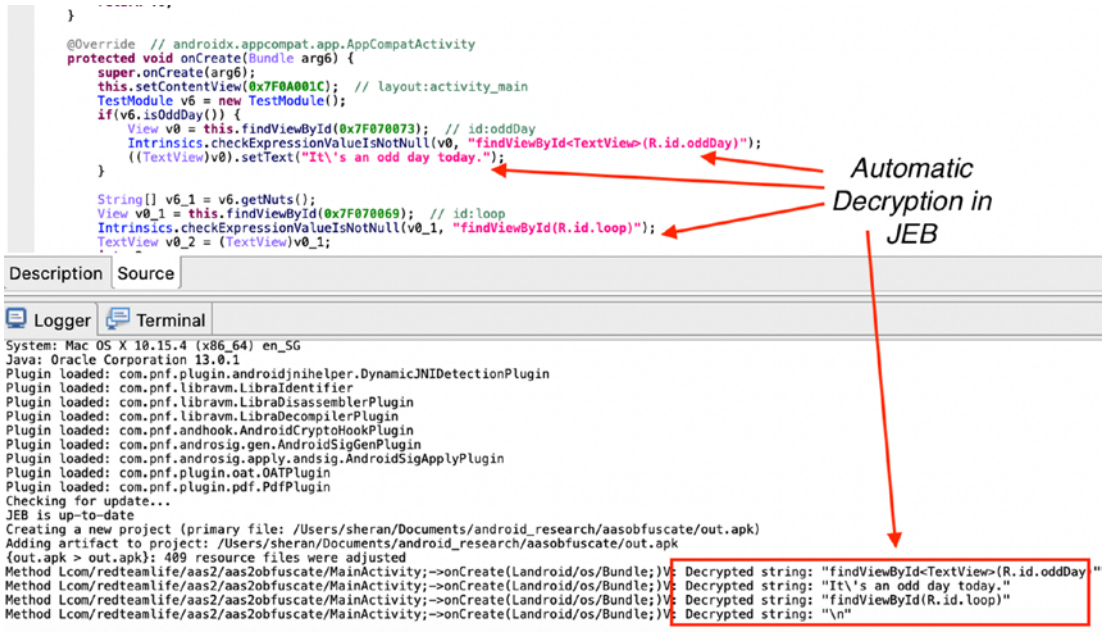


Figure 4-7. *Showing automatic string decryption in JEB*

It is important to keep in mind that the quality of the obfuscator you use will obviously yield better results at foiling reverse engineers. It also goes without saying that the more higher-quality obfuscators cost a lot of money to purchase and use. Here are a few examples of obfuscation techniques found in apps from well-funded startups. They can definitely afford to shell out the money to protect their apps.

Another example of class naming is shown in Figure 4-8.

Figure 4-8. *Class renaming example*

Here is an example of hiding a string within a method whose only purpose is to return a string:

```
01:  static void a(Context arg6, Hellfire arg7) {
02:         String v2_1;
03:         boolean v1_3;
04:         b v1_2;
05:         com.component.secure.hellfire.d.b.b v1_1;
06:         String v5;
07:         a v0;
08:         try {
09:             c.c.lock();
10:             v0 = a.c();
11:             if(v0.a().needNtpSync()) {
12:                 HashMap v1 = new HashMap();
13:                 v1.put("X-Gxxx-Exx", "gxxx-exx");
14:                 v5 = null;
```

```
15:                 v1_1 = com.component.secure.hellfire.d.b.a(new com.
                    component.secure.hellfire.d.a(b.a(), com.component.
                    secure.hellfire.d.b.a.GET.method, null, v1));
16:                 int v2 = c.a.a[v1_1.a.ordinal()];
17:                 if(v2 == 1) {
18:                     goto label_63;
19:                 }
20:
21:                 if(v2 != 2) {
22:                     if(v2 != 3) {
23:                         goto label_38;
24:                     }
25:
26:                     v0.a(com.component.secure.hellfire.a.a.DISABLED,
                        v1_1.b);
27:                     goto label_74;
```

The method `b.a()` is the one that returns the string. You can see it in lines 22–24 as follows:

```
01:         ...
02:
03:         ...
04:
05:         ...
06:         catch(KeyManagementException v5_1) {
07:             v0.a = com.component.secure.hellfire.d.b.b.a.FAILURE;
08:             v0.b = v0.b + v5_1.getMessage();
09:             return v0;
10:         }
11:         catch(Throwable v5) {
12:             throw v5;
13:         }
14:
```

```
15:         if(v2 != null) {
16:             v2.disconnect();
17:         }
18:
19:         return v0;
20:     }
21:
22:     public static String a() {
23:         return "https://api.xxxx.xxx/gxx/v1/lxxxxx";
24:     }
25: }
```

Here is an example of using nonalphabetic characters to name classes (Figure 4-9).

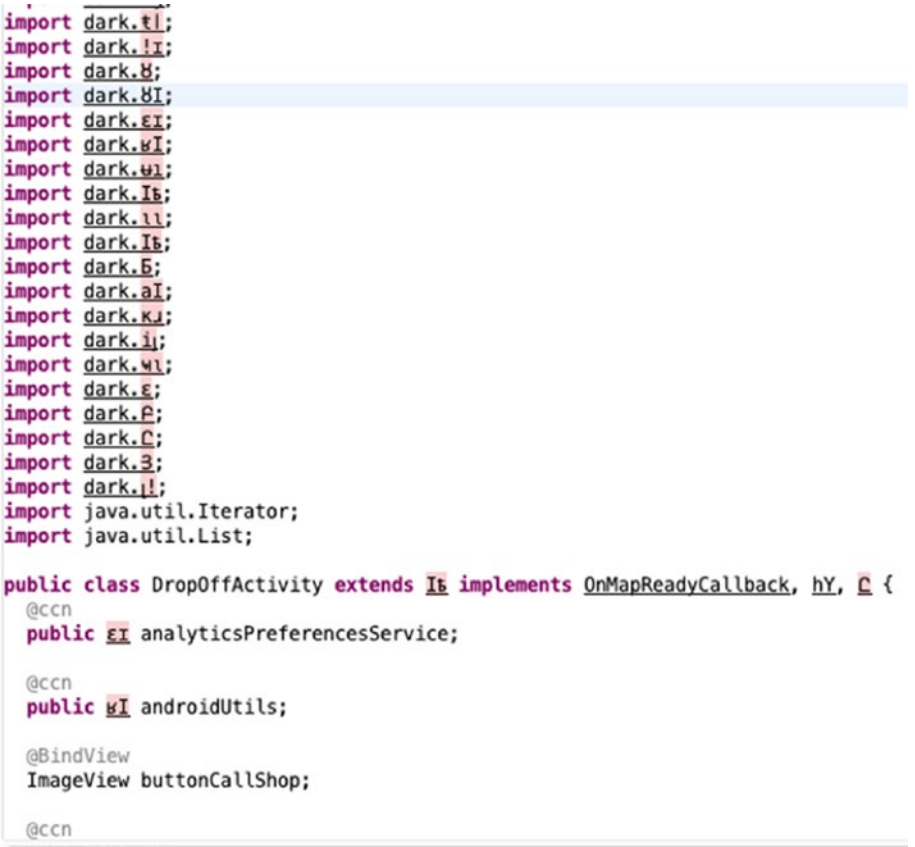


Figure 4-9. *Example of obfuscation renaming classes to nonalphabetic characters*

Here is an example of using weird file names when you unzip the APK file (Figure 4-10).

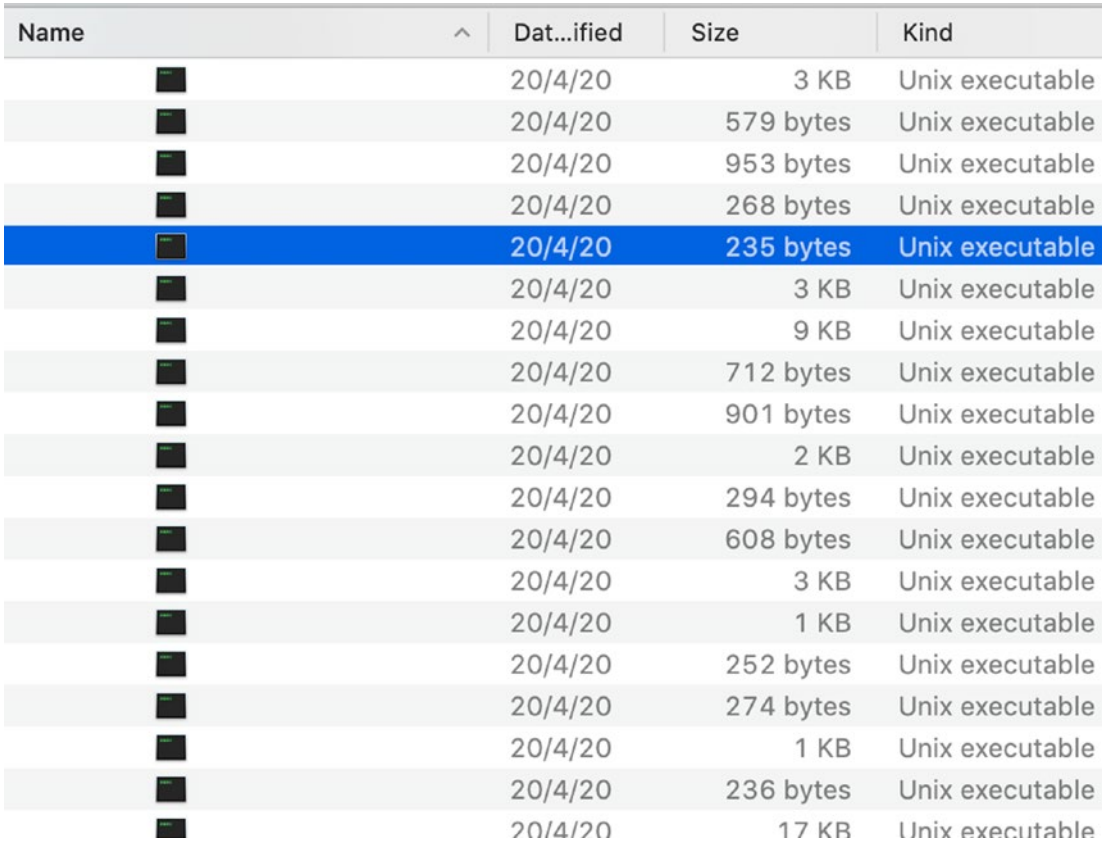


Figure 4-10. *Empty file names (using nonprintable ASCII characters) when an APK is unzipped*

Summary

There are many ways and many tools that you can use to obfuscate your code. The most effective ones can greatly slow down a run-of-the-mill reverse engineer who is out looking for low-hanging fruit. The more determined, competent ones, however, will be able to make some sense of your app enough to either modify it or understand what it does to further his attack. If you must use obfuscation, do so. But know that it is not something that can make your code invisible to your attacker.

Vulnerability Assessment

Depending on the size of your company, you may or may not have an in-house security team. If you are lucky enough to have an in-house offensive security team, you should not waste the opportunity to incorporate them into the entire development life cycle. Again, with this one, there's plenty of ways you can structure how you want your red and blue teams involved in your software development process. I'll take a moment to tell you how I structured the team that I had and the roles of each of these teams and their tasks. I will tell you what I hoped to achieve and what I really achieved. With hindsight, I also have some thoughts on what I think I should have improved upon and how I should have done things differently. Perhaps you can glean some knowledge from my experience and come up with your own process and improve upon it. If you do and find something that works well for you, do drop me an email and tell me how you're getting on. I am always keen to learn.

The Red Team

The Red Team is your offensive team. They break and break into things – generally your own things. A Red Team's job is to take stock of all assets that belong to the company and then systematically attack them. The attacks can be controlled or ad hoc. By this, I mean you can give your Red Team a general mandate and let them run loose. This means that all attacks they perform are at their discretion, and no one knows when an attack will take place. For me, I chose the more controlled route and set the team up as an important part of the software development cycle. We met with many of the development teams that would be responsible for releasing a product that went inside the app (the app itself was more like a container that held many different services; what you would categorize as a super app) and told them that any new functionality that was rolled out would have to have a mandatory vulnerability assessment performed. Any high- or medium-risk findings from this assessment would have to be rectified before the app could be considered for release. By communicating this to the development teams and, more importantly, enforcing it, we saw a shift in how the development process changed. We began to see more planning being done on the development side to incorporate time required for the security assessment, fixing, and retest. Of course, there were always exceptions and teams that didn't want to follow the process. We took these in stride and did our best to educate the teams and have an open dialogue with them. Ultimately, it is important to have a dispute resolution process where the decision

making on whether to launch or not launch lies with the senior leadership. Much like a court case where a jury hears both sides before making a decision can work wonders. What is important is to have on record that leadership has chosen to proceed with the solution and is aware of the trade-offs of picking one over the other.

Another area that the Red Team would focus on was research. They would be the team to stay informed of the latest security threats, make proof of concepts, and attack our own internal systems to see if they were vulnerable rather than waiting for those exploits to make it to the tools that they used.

The Blue Team

The Blue Team takes on the opposing role to the Red Team. They are all about defense. The Blue Team will make sure that some of the following areas are secure:

- Cloud computing infrastructure: This means who has access to the cloud resources, who can see billing information, who gets to provision new billable services, and so on.
- Gatekeeping: Controlling access to firewall ports and load balancer APIs.

The cloud infrastructure component should be self-explanatory, but I wanted to elaborate a little on the gatekeeping aspect of the Blue Team. If you remember, there was a process in place that if you were a product team that wished to deploy a new feature, then you would need to have a vulnerability assessment done first. One of the checks that the Blue Team would make before they put that service online and into production was to ask that question and review the results of the vulnerability assessment report. So sometimes if a team decided to "fast track" their code into production, they would often have to go back and get an assessment done first.

A Word About Automation

The Red Team and Blue Team often had a lot of repetitive tasks. For example, the Blue Team would need to check on all firewall ports and APIs exposed through the load balancing framework. If this were to be checked by hand, then it would waste a lot of the engineers' time. So this was automated to have a daily report sent in to the team if there were any deviations from the baseline. Similarly, other checks on the cloud infrastructure were also automated. The Red Team, likewise, had their areas of

automation. For example, they would be able to script a large part of their vulnerability assessment to get all the repeatable tasks out of the way, and they would then only need to focus on the more dynamic or more variable parts of the test – an instance where they could feed in an APK to their automation framework and the framework would decompress and map out all functions or calls to the back-end server that the team could immediately pick up and start testing rather than having to go through a dead listing of static, disassembled code.

The Compliance Team

The Compliance Team would be the team that built, maintained, and enforced policies within the organization. All relevant security policies would originate from the Compliance Team and be communicated organization-wide. Then, regular audits would be performed by the team to ensure that policies were being followed. Further, the Compliance Team would be the link between the legal team and the country's regulatory bodies such as the central bank or other authority that demanded that the company comply with prevailing laws and regulations. The team would facilitate external audits from such regulatory bodies as well.

Visualizing the Team

If I were to visualize what the team would look like within the engineering space of the organization, I think it would look something like Figure 4-11.

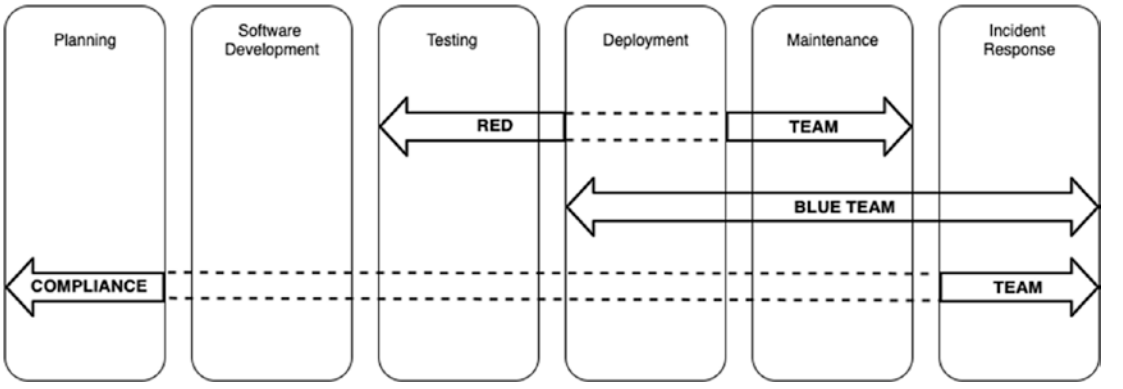


Figure 4-11. *The teams fitting into the engineering space of the organization. Dotted lines indicate that the team does not handle that process.*

Looking through the visualization, I can immediately see that the Planning and Development stages of engineering could use some help. Further, I think I could extend the Red Team to help with the Incident Response as well. I could use the source code analysis on the Software Development component for sure and place that under the Blue Team.

Improvements

As promised, I think if I were to look back on the team setup, I would put a lot more effort into the following:

- Communications: We did not communicate nearly enough with other teams, and this could have been improved. Further, we didn't evangelize and educate as much as we could, especially in a more inclusive manner.
- Attitude: We could have been more empathetic toward the engineering team in general. I think we had a little bit too much of a strong attitude to render us a little unapproachable. Patience does wear thin, but I think I could have led by example in this case.
- Securing leadership buy-in: This is critical. We did have a lot of support from the leadership. We could have, however, spent a lot more time in constant communication with the leadership so as to retain security front and center.

Looking back, we were a team that was more technical and real-world attack oriented. This may be a good thing, yet it left a gap in the relationship building and educational areas of the cause. If you find yourself in this dilemma, then the best thing you can do for yourself and your team is to get an "interpreter." Typically, this could be something akin to a product manager – a person who is on the frontlines, building the reputation and relationships of the security team. Include one or two of these people in your hiring along with your seasoned hackers.

Running on the Emulator

Some companies will make it even harder for reverse engineers. Generally, the easiest place to debug, reverse, or test APKs is the Android emulator. It's easy to fire up and even easier to get root access on. So many security researchers and attackers alike use it widely.

Because of its lax nature, some companies will try to take steps to ensure that APKs do not run on the emulator. Some ways in which I have seen them achieve this is to look for hardware-specific components. One is looking for the Bluetooth adapter like this:

```
01: val btAdapter = BluetoothAdapter.getDefaultAdapter()
02: Log.d("aas2obfuscate","BT Adapter address is "+ btAdapter.address)
```

If successful, your app should work as planned; if it is the Android emulator, then the app will crash, throwing an error like this:

```
E/BluetoothAdapter: Bluetooth binder is null
D/AndroidRuntime: Shutting down VM
E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.redteamlife.aas2.aas2obfuscate, PID: 31179
    java.lang.RuntimeException: Unable to start activity ComponentInfo{com.
    redteamlife.aas2.aas2obfuscate/com.redteamlife.aas2.aas2obfuscate.
    MainActivity}: java.lang.IllegalStateException: btAdapter must not be null
        at android.app.ActivityThread.performLaunchActivity(ActivityThread.
        java:2817)
        at android.app.ActivityThread.handleLaunchActivity(ActivityThread.
        java:2892)
        at android.app.ActivityThread.-wrap11(Unknown Source:0)
        at android.app.ActivityThread$H.handleMessage(ActivityThread.
        java:1593)
...
...
```

There is a third-party library which is a little old, but may still prove to be useful at detecting if an emulator is present. You can find the source code here: https://github.com/daiwei92/android-emulator-detector. According to the author, when checked in 2018, the library was able to detect the following emulators:

- Checked on real devices in Device Farm (https://aws.amazon.com/device-farm/)
- BlueStacks
- Genymotion
- Android Emulator

- Andy 46.2.207.0
- MEmu play
- Nox App Player
- KoPlayer

When trying to detect if an emulator is present, one way in which you can do it is to check for the device's IMEI number. Android emulators don't have this number so the check should fail. The drawback to this is that the permission `READ_PHONE_STATE` is quite likely that a user will most likely not provide this permission out of fear that the app is trying to record his call logs or worse. Thus, you will have to use caution when using this technique.

CHAPTER 5

Hacking Your App

In this chapter, I would like to talk about some steps you can take to testing the security of your own app. In a larger organization or a more mature startup (if there is such a thing), the testing may usually be done by the information security team. But before we get started on the technical aspects of hacking an Android app, I want to take a moment to talk about the security testing process.

As is normal, in a startup, leadership roles rarely blend. They are distinct and have a purpose. Now while the goal of the startup may be to achieve tremendous growth in user base, or ultimately to IPO, it is the collective paths and efforts of all the employees toward that goal that will result in its successful attainment. Well, in theory at least; because for this to work smoothly, you have to assume that everyone has almost no ego, look out for the good of the company, and are more likely to pick the company goals as a higher priority. For this discussion, let's assume that this is the case.

I bring up the leadership roles as being distinct and purposeful for a reason. Most times, because of the purposefulness of the role, it sometimes feels like other leaders are preventing you from doing your part as you strive toward the common goal. I will illustrate this with a very simple example.

The CTO has had his engineering team deliver a new set of features that affect the payment component of your app. Let's assume that the feature allows you to now transfer money from your wallet to a friend's wallet. We can all agree that many a startup has achieved greatness on that sole feature alone, so we know it's a great feature. The CTO and his team are happy, and they talk about it with the CEO and he's happy, and in general there is an air of pride, joy, and success. In the midst of this celebratory high fiving, the CISO walks into the room, pulls the CTO aside, and says, "I think we have a problem. My team found a flaw in the feature where many users can potentially lose a lot of money. We think that the solution for this would be to implement a few safety mechanisms prior to the user transferring the money. It is safer, and ultimately, we protect our users."

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_5

What would your reaction be as the CTO? How would you characterize how the CISO felt? Taking a high-altitude look at this problem, and because I have set the tone of this example, it is easy to see that both the CTO and the CISO are keen to advance their team contributions toward furthering the company goal of becoming successful. The CTO just built out a stellar feature that would attract and retain many users onto the app platform. The CISO wants to ensure that the users trust and remain on the platform rather than realizing that the app is unsafe and could cause them or their friends to lose money.

How does this usually get resolved if neither the CTO nor the CISO is willing to compromise their position? Well usually, it gets escalated to the CEO, or better still it is put toward a vote among the leadership. This is where things get interesting. This is the point when you learn whether information security is a top-down mandate or whether growth at all costs is the mandate. In some cases, things may not be so binary, but decisions made at this moment in a startup's life cycle are usually repeated unless a breach has occurred between such decisions. In cases like that, things become more interesting with either an overcorrection on information security or a slowdown in growth, but we won't cover that situation here. Regardless of how this turns out, I wanted to point out the number of stakeholders involved in a situation like this. If you're a CISO in this position, like how I was many times in the past, you would do well to consider the viewpoints of all other parties involved and work toward finding a solution. I think the one key thing a CISO has to learn even more than the technology part is the communications part. A CISO has to be masterful at communicating, persuading, educating, and harmonizing teams.

The situation I describe here assumes that no formal security testing process has been fixed nor communicated to all involved. This is usually the case because the status quo of a startup has most likely been to build and push forward and grow at all costs. It is worth diving into this part a little closely as it is an important discussion to have beyond just the technical parts of what a security team does.

For the rest of this chapter, let's assume that we are part of that team that has to formally test apps before they go live. It's your first day on the job, and you're getting stuck into your testing environment. I will take you through some of the steps you may encounter in that role and discuss them in a little bit more detail. Going forward from here, let's think that we have in our possession the app that we will be checking the security of. Let's go on from there.

Feature Examination

The first thing you're going to want to do is use the app and understand what its features are. Sometimes it might be difficult because it may be a closed system and require you to preregister or be part of the organization that releases the app, in which case you can move on to the next part of this section.

One of the things I do is to check how the app looks; after years of looking at apps, you develop a sense for whether an app is natively written on or one written in HTML5 or React Native. This is so that I can take a mental note of some of the things I want to check in the next stage. I then look for interesting areas within the app that I want to focus my efforts on. This could be a login or password entry screen, a shopping cart or purchase screen, or a screen that returns data to the user like a user profile screen. When we go deeper in other chapters, we will cover some of the differences between native and HTML5, but for now let's leave that alone.

Getting the APK File

Now that we have looked through the features of the app and we have made notes on which areas to examine, we can get started collecting the APK. The APK is the Android package file that contains all the working elements necessary to run the app on an Android device. It's actually a ZIP file format that you can easily decompress to view the contents, as you can see in Figure 5-1.

```
→  test ls
connected.apk
→  test unzip -q connected.apk
→  test ls
AndroidManifest.xml                      org
META-INF                                 places.properties
android-logger.properties                play-services-ads-identifier.properties
androidsupportmultidexversion.txt        play-services-base.properties
assets                                   play-services-basement.properties
classes.dex                              play-services-clearcut.properties
classes2.dex                             play-services-location.properties
classes3.dex                             play-services-maps.properties
com                                      play-services-measurement-api.properties
connected.apk                            play-services-measurement-base.properties
firebase-analytics.properties            play-services-measurement-impl.properties
firebase-common.properties               play-services-measurement-sdk-api.properties
firebase-core.properties                 play-services-measurement-sdk.properties
firebase-iid-interop.properties          play-services-measurement.properties
firebase-iid.properties                  play-services-phenotype.properties
firebase-measurement-connector.properties play-services-places-placereport.properties
firebase-messaging.properties            play-services-stats.properties
kotlin                                   play-services-tasks.properties
lib                                      res
okhttp3                                  resources.arsc
→  test
```

Figure 5-1. *Running unzip on an APK file*

Generally, if you want to poke around under the hood very superficially, you can use the unzip trick and see if there's anything that catches your interest that you can further pursue. But we're not here to mess about with random apps downloaded from the Play Store; we're here to hack our own apps, remember? OK, let's move on.

The Android Debug Bridge (adb)

In a previous chapter, we looked at Android Studio. This is an IDE that developers use to write, build, and deploy Android apps. We now have to go back to Android Studio and install some platform tools. Platform tools give us some additional tools, one of which is the Android Debug Bridge (adb) as you will see; this tool is frequently used and is very useful when both developing and hacking Android apps. In Android Studio's welcome screen, click Configure and then click SDK Manager, then click SDK Tools, and tick the box for Android SDK Platform-Tools as shown in Figure 5-2.

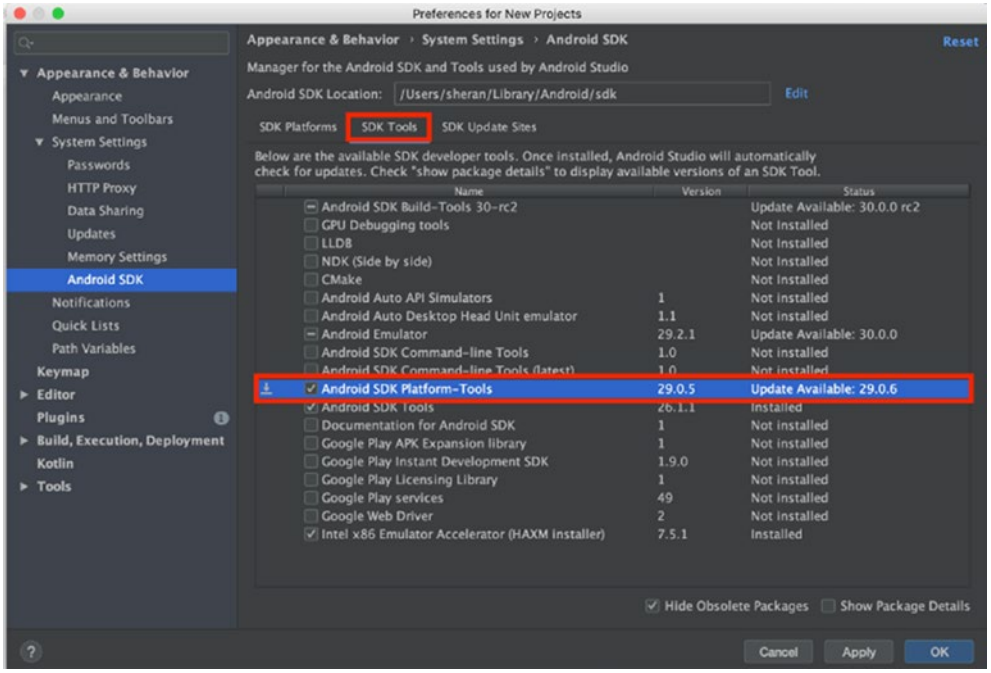


Figure 5-2. *Installing Android SDK Platform-Tools on Android Studio*

Then click the OK button, and after answering yes to the question whether you want to proceed with the installation, the platform tools should be installed. You can see the location to which Android Studio installs the platform tools in the installation progress window, part of which is shown in Figure 5-3.

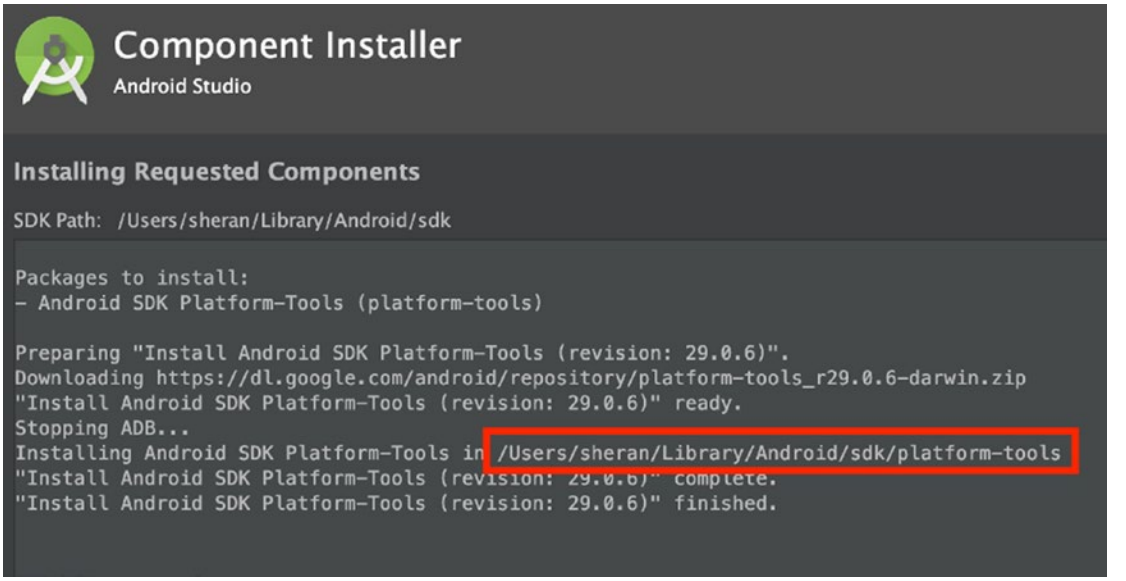


Figure 5-3. *The platform tools installation progress and installation location*

When complete, click Finish and then we should have adb installed. One thing you will need to do if you use adb frequently is to add the location of adb to your path. On MacOS, this is easy enough. Open your terminal and then change directory to `/etc/paths.d/`

```
cd /etc/paths.d
```

Then let's create a file called adb that will contain our path to the platform tools location. In my case, running Android Studio 3.6.3, the location is `/Users/sheran/Library/Android/sdk/platform-tools`

```
sudo nano adb
```

Enter your password when prompted. Now type in or copy and paste your location into the file. When you're done, press `Ctrl + x` to exit. You will be prompted whether you want to save the file, so press `Y`. Now if you list the contents of your `/etc/paths.d/` directory, you will see the file that you newly created like in Figure 5-4.

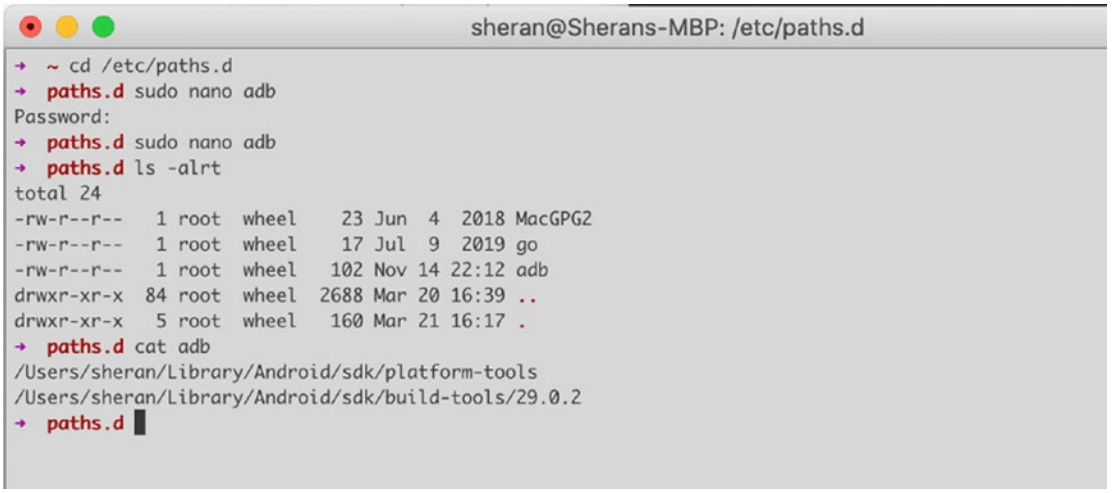


Figure 5-4. *Creating the adb file to add the platform tools to the path*

Then close and reopen your terminal to start a new shell so that your new paths are in place and type in adb. You should see a response similar to Figure 5-5. The help and various parameters for adb go on for a few pages, but I have captured just the first few lines for brevity. If your adb doesn't work or you get an `adb: command not found` response, then you haven't set your path up correctly, or you haven't installed the platform tools correctly. While I would take great joy in helping you figure those out to further fatten up my book, I shall not. Instead, be good little ones and go ask Google for some help. I'm here all week. I'll wait.

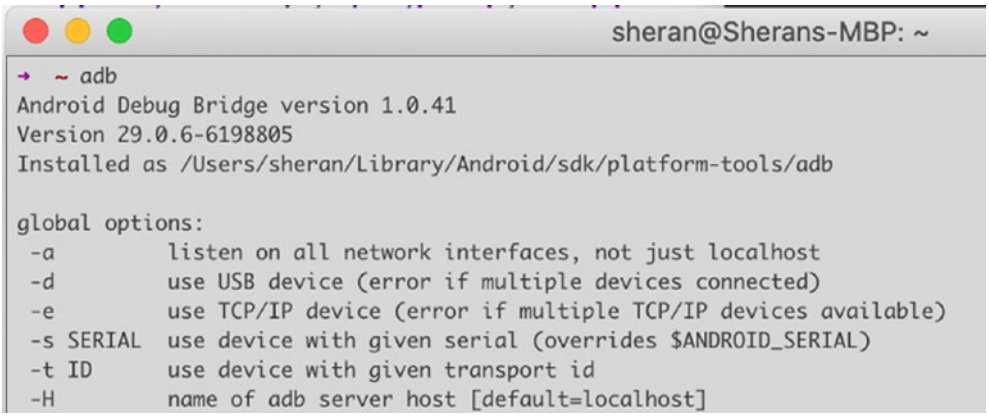


Figure 5-5. *Output from adb when installed correctly*

Throughout this book, I keep looking for a glorious opportunity to write "Now here's where the rubber really meets the road!" I thought this would be it, but didn't want to squander an opportunity, so just hold that phrase in your head for now. First, let's talk a little bit about the Android Debug Bridge and how it connects to your phone. The Android Debug Bridge allows you to debug apps and communicate with an Android device from the command line. It can provide you with a Unix shell on the device so that you can run further commands from the context of the device. The entire adb framework comprises of three parts:

1. The adb client: This is what you will always use when you want to install an app onto a device, copy an apk from a device to your workstation, or invoke a remote shell on a device. This runs on your workstation.
2. The adb server: This part is always running on your workstation. In some cases, you may also have to start the server manually by entering `adb start-server`. The first time you invoke the adb command, you may see a message that says `* daemon not running; starting now at tcp:5037`. This is the adb server starting up on your workstation. Figure 5-6 shows the output of the `lsof` command showing a listening adb server.
3. The adb daemon: This part runs on every Android device. Also running in the background, the adb device daemon is always running and ready to connect with a server and execute client commands as necessary.

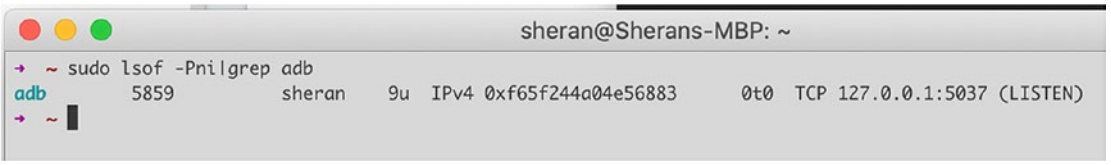


Figure 5-6. *Output of the lsof command that shows a listening adb server*

The client talks to the server which then makes a connection to the remote Android device's adb daemon. It looks like Figure 5-7.

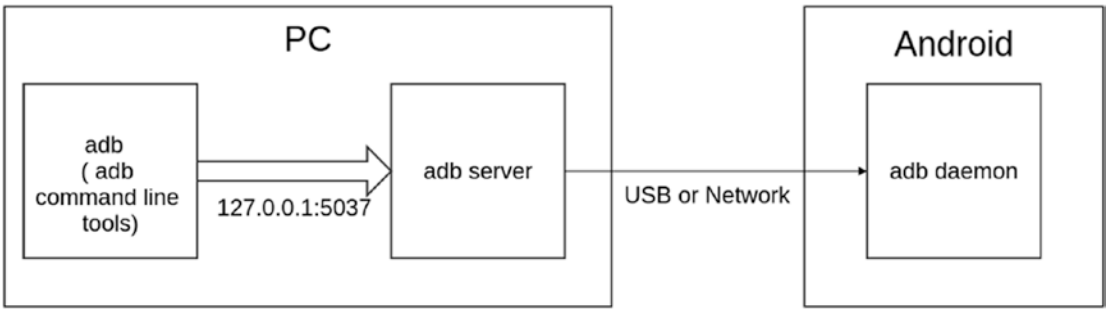


Figure 5-7. *The three components of the adb framework*

Developer Mode

You may be concerned that your Android device is always running around with a daemon or server listening for all manner of connections from either USB or the network, but it isn't that simple. First, you have to put your phone into Developer Mode and then enable USB debugging. This is the stage when your phone will listen for commands coming from a remote adb server.

The way to put your device into Developer Mode is quite straightforward.

1. Open your Android device's Settings screen.
2. Scroll down to About phone and click it.
3. Locate the item that says Build number and keep tapping it until a little countdown tells you how many more taps of your finger are needed until you are a developer.

Yes, I know, exciting stuff! Figures 5-8 and 5-9 show what it looks like when you're already a developer on two different phones.

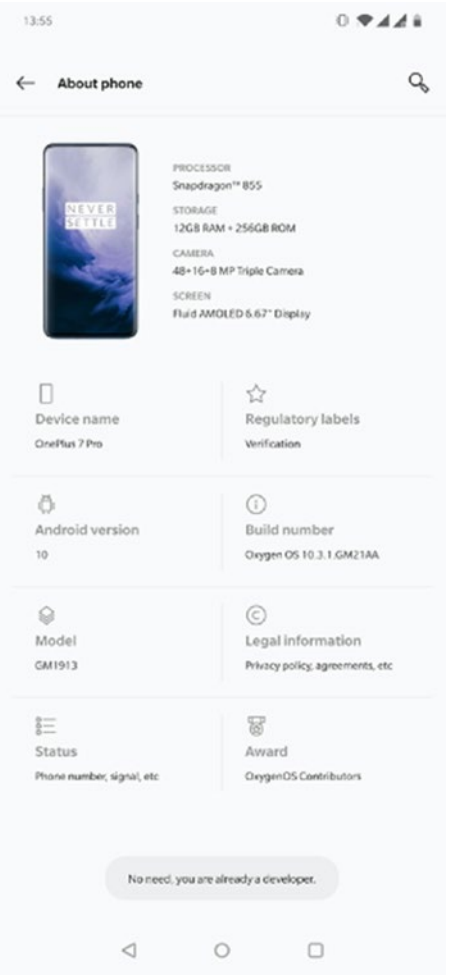


Figure 5-8. *Becoming a developer on a OnePlus 7 Pro*

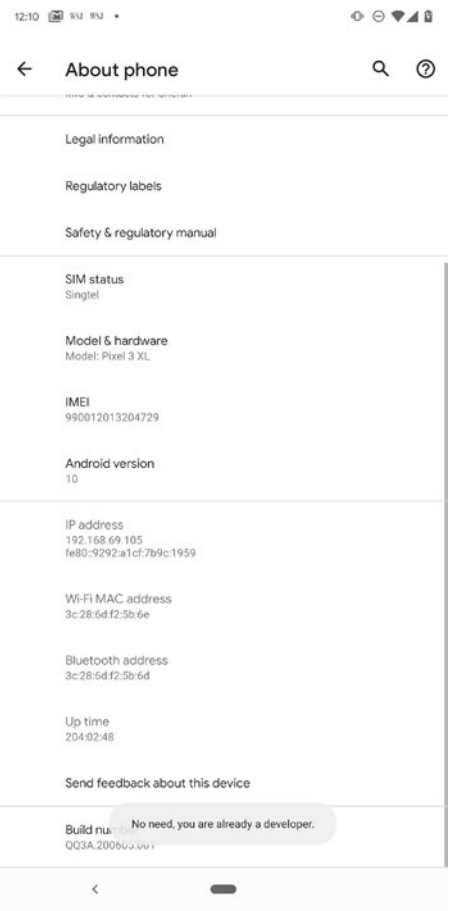


Figure 5-9. *Becoming a developer on a Google Pixel 3 XL*

By becoming a developer, you get access to a hidden menu under the System menu of the device Settings menu. Now go back to the System menu, and you will see Developer Options. Further, within the Developer Options menu, scroll down to where the title says Debugging, and look for the USB debugging switch and turn it on. You should get a confirmation dialog box asking you to confirm. Press OK and then USB debugging mode will be switched on!

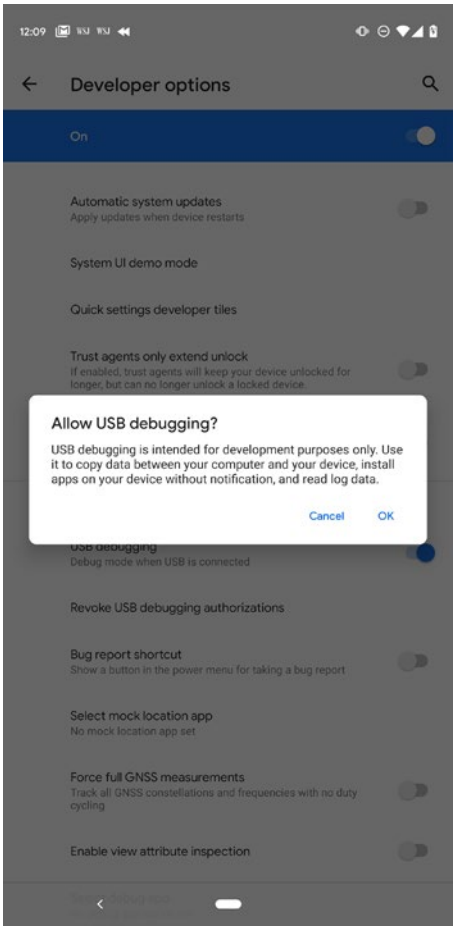


Figure 5-10. *Screenshot of the message you get when you enable USB debugging on a Google Pixel*

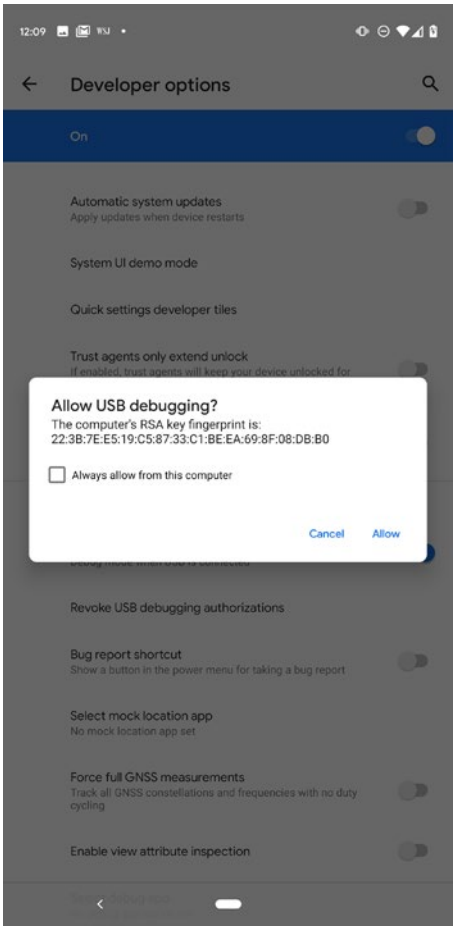


Figure 5-11. *Prompt to trust the computer that you just connected*

Super! Now let's connect our phone to our workstation, shall we? Grab a hold of your USB cable that came with your phone and plug the phone into your workstation. In my case, I will plug it into my MacBook. One useful and secure feature of the newer Android builds is that whenever you plug the phone into a computer, you get prompted whether to trust the device or not. This makes things a lot safer from folks who think they could just rock up to you at a Starbucks when your back is turned, plug their computer into your phone and instantly root your device, hack it, and walk away. If you don't keep your phone locked, they can still do this because they can just allow access themselves. But this will never happen because we all protect our phones with a password or biometric access, right? Good. For now, go ahead and trust your own computer by clicking Allow (Figure 5-11).

Next, let's see if we can find our device using the command line and adb. The first thing we want to do is list out the devices that adb can see. Type in the following and press enter:

```
adb devices
```

That should result in an output similar to that in Figure 5-12.

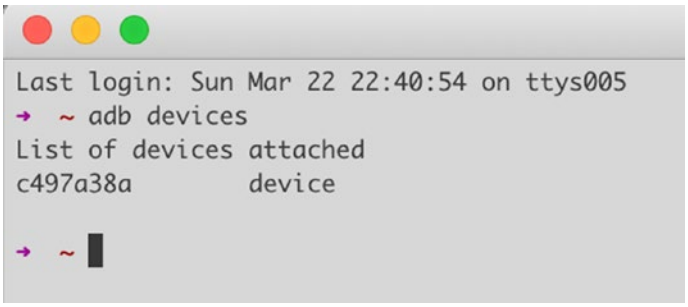


Figure 5-12. *Listing of adb devices*

Let's open up a shell on our phones using adb. Type in the following:

```
adb -d shell
```

This should open up a Unix shell directly on your device. Now you're free to navigate through your device as you would a similar Unix terminal. This is what I see when I do an `ls -al` on my OnePlus 7 Pro:

```
1|OnePlus7Pro:/ $ ls -al
ls: ./init.zygote64_32.rc: Permission denied
ls: ./init.rc: Permission denied
ls: ./init.usb.rc: Permission denied
ls: ./ueventd.rc: Permission denied
ls: ./op2: Permission denied
ls: ./init.zygote32.rc: Permission denied
ls: ./init: Permission denied
ls: ./cache: Permission denied
ls: ./init.oem_ftm.rc: Permission denied
ls: ./init.environ.rc: Permission denied
ls: ./init.recovery.qcom.rc: Permission denied
ls: ./postinstall: Permission denied
ls: ./init.usb.configfs.rc: Permission denied
```

```
ls: ./op1: Permission denied
ls: ./metadata: Permission denied
total 64
drwxr-xr-x  24 root   root    4096 2009-01-01 08:00 .
drwxr-xr-x  24 root   root    4096 2009-01-01 08:00 ..
dr-xr-xr-x 967 root   root    0 1970-09-18 17:10 acct
drwxr-xr-x  17 root   root     340 1970-09-18 17:10 apex
lrw-r--r--   1 root   root     11 2009-01-01 08:00 bin -> /system/bin
lrw-r--r--   1 root   root     50 2009-01-01 08:00 bugreports -> /data/
                               user_de/0/com.android.shell/files/bugreports
lrw-r--r--   1 root   root     19 2009-01-01 08:00 charger -> /system/bin/
                               charger
lrw-r--r--   1 root   root     17 2009-01-01 08:00 charger_log -> /sbin/
                               charger_log
drwxr-xr-x   5 root   root     0 1970-01-01 07:30 config
lrw-r--r--   1 root   root      17 2009-01-01 08:00 d -> /sys/kernel/debug
drwxrwx--x  59 system system   4096 1970-09-10 06:19 data
drwxr-xr-x   2 root   root     4096 2009-01-01 08:00 debug_ramdisk
lrw-------   1 root   root     23 2009-01-01 08:00 default.prop -> system/
                               etc/prop.default
drwxr-xr-x  18 root   root     5780 2020-03-02 18:45 dev
lrw-r--r--   1 root   root      11 2009-01-01 08:00 etc -> /system/etc
drwx------   2 root   root     16384 2009-01-01 08:00 lost+found
drwxr-xr-x  12 root   system   260 1970-09-18 17:10 mnt
drwxr-xr-x   4 root   root    4096 2009-01-01 08:00 odm
drwxr-xr-x   2 root   root    4096 2009-01-01 08:00 oem
drwxr-xr-x   2 root   root    4096 2009-01-01 08:00 oneplus
dr-xr-xr-x 823 root   root       0 1970-01-01 07:30 proc
lrw-r--r--   1 root   root      15 2009-01-01 08:00 product -> /system/product
lrw-r--r--   1 root   root      24 2009-01-01 08:00 product_services ->
                                /system/product_services
drwxr-xr-x   3 root   root    4096 2009-01-01 08:00 res
drwxr-x---   2 root   shell   4096 2009-01-01 08:00 sbin
lrw-r--r--   1 root   root      21 2009-01-01 08:00 sdcard -> /storage/
                                self/primary
drwxr-xr-x   4 root   root      80 2020-03-02 18:45 storage
```

```
dr-xr-xr-x  16 root   root       0 1970-09-18 17:10 sys
drwxr-xr-x  18 root   root    4096 2009-01-01 08:00 system
drwxr-xr-x  16 root   shell   4096 2009-01-01 08:00 vendor
1|OnePlus7Pro:/ $
```

Another useful tool that you can run through the adb shell on your device is the package manager or pm. The package manager allows you to collect a lot of information about packages, modules, and libraries and even set permissions on a granular level. Let's use the pm utility to get a list of all the packages installed on our device. Go ahead and run the following command:

```
pm list packages
```

When I run it on my device, I get the following output:

```
255|marlin:/ $ pm list packages
package:com.google.android.carriersetup
package:com.android.cts.priv.ctsshim
package:com.google.android.youtube
package:com.vzw.apnlib
package:com.android.internal.display.cutout.emulation.corner
package:com.google.android.ext.services
package:com.android.internal.display.cutout.emulation.double
package:com.android.providers.telephony
package:com.android.dynsystem
package:com.android.sdm.plugins.connmo
package:com.google.android.googlequicksearchbox
package:com.android.providers.calendar
package:com.android.providers.media
package:com.google.android.apps.docs.editors.docs
package:org.proxydroid
package:com.qti.service.colorservice
package:com.android.theme.icon.square
package:com.google.android.onetimeinitializer
package:com.google.android.ext.shared
package:com.android.internal.systemui.navbar.gestural_wide_back
package:com.qualcomm.ltebc_vzw
package:com.qualcomm.shutdownlistner
```

```
package:com.quicinc.cne.CNEService
package:com.android.theme.color.cinnamon
package:com.htc.omadm.trigger
package:com.android.theme.icon_pack.rounded.systemui
package:com.android.documentsui
package:com.android.externalstorage
```

Let's assume the app that we want to download off of the phone is the Lylt app. This is an app that we developed at the company I work in. You can find it on the Google Play Store. We can see that the package name is `package:com.lylt.customer` which is something we need to take note of. In your terminal, make sure you are in a working directory where you can store the apk and continue to work with it. I usually have something like `work/apks/<package>/<package files>` so that I can individually store the apks and then either make copies of them or unpack and repack them as required. It's neater. To copy a package or apk from an Android device to your computer, you will also need to know where the package files are located. You can do this again with the `pm path [PACKAGE]` command like so:

```
255|OnePlus7Pro:/ $ pm path com.lylt.customer
package:/data/app/com.lylt.customer-gHnL6VGqnnsi5YT6GMRRIg==/base.apk
OnePlus7Pro:/ $
```

Let's go over what we did. We took the package name, which is `com.lylt.customer` (notice we stripped out the word "package:" from when the `pm list` command gave us all the package names). We used the path command with pm this time so that it will tell us exactly where on the device the apk is installed. In this case, the apk is located at `/data/app/com.lylt.customer-gHnL6VGqnnsi5YT6GMRRIg==/base.apk`. Now to copy this back to our device, we will have to run this command from adb and not pm. Type `exit` to leave the device's Unix shell. Make sure you're in your correct working directory and then look through the following sequence of commands and responses:

```
➜  ~ adb pull /data/app/com.lylt.customer-gHnL6VGqnnsi5YT6GMRRIg==/base.
    apk lylt.apk
/data/app/com.lylt.customer-gHnL6VGqnnsi5YT6GMRRIg==/base....le pulled, 0
skipped. 36.8 MB/s (10000865 bytes in 0.259s)
➜  ~
```

The command itself is straightforward. We use `adb pull` to pull the apk from the device. We supply the `adb pull` command with our path. Then we leave a space and give it the name `lylt.apk`. In Android when apps are installed, they are named `base.apk` which gets a bit confusing when you have to work with many of them, so I rename it whenever I pull the apk to my computer – which is what we have done.

Our next step is to look inside the apk and figure out what is going on in there. There are a few different ways of doing this and also different tools that we can use. I will talk about how I analyze an apk and the tools that I use.

Static Analysis

Static analysis is one of the ways you can look at an app. The term static means that the app isn't running. Some of the typical static analysis techniques are reverse engineering which fall into either decompiling or disassembling. Decompiling means you reverse a binary, compiled version of the app back into its source code, and disassembling means you convert the binary to a possibly less human-readable code more closely matching machine language. Typically, this is known as the assembly language, hence disassembly. With Android apps, we usually write them in Java or Kotlin. In some cases, we may use C or C++ to build parts of an app that require more compact and performant code than what either Java or Kotlin can provide. Imagine that if an Android app is written in Java or Kotlin, compiled, and then distributed on the app store, you already know how to install it on your phone and then copy it to your computer so you can do some static analysis on it. If we decompile it, we are hoping to get back Java or Kotlin code so we can better decipher how the app was built.

The go-to tool of mine is a commercial decompiler for Android apps called JEB. It is written and maintained by PNF Software which I'm sure for the most part was largely developed by its founder Nicolas Falliere. It is a very nice piece of software that is very self-contained and can even do dynamic analysis. You can extend it by writing your own scripts to deal with some repeating part of the analysis you want to do, or you can write a customized deobfuscator (more on that later). But I would be remiss in my duties as a knowledge sharer if I didn't talk about a free solution as well. Thus, before we move on to JEB, let's look at APKTool.

APKTool

APKTool is a command-line tool for reverse engineering Android apps. You can find APKTool at https://ibotpeaches.github.io/Apktool/. Getting APKTool is very easy, especially if you don't want to build it from source. The instructions are quite simple, and I leave it to you as an exercise to follow from here: https://ibotpeaches.github.io/Apktool/install/.

If you're a Mac user, you can easily install APKTool from Homebrew by executing

```
brew install apktool
```

APKTool does not give you full Java or Kotlin back from a decoding session. It doesn't give you assembly or Java bytecode either. What it gives you is smali code. Smali is a hybrid between bytecode and full-blown Java. It sits closer to the bytecode spectrum, though. Smali and baksmali were created by Ben Gruver, who goes by the more easily recognizable handle of JesusFreke. They are an assembler and disassembler for dex files that were used by Android's version of the Java Virtual Machine, named Dalvik. I will let JesusFreke himself describe the naming of smali and baksmali:

> *The names "smali" and "baksmali" are the Icelandic equivalents of "assembler" and "disassembler" respectively. Why Icelandic you ask? Because dalvik was named for an Icelandic fishing village.*

After you've got APKTool up and running, we can then reverse engineer the Lylt app that we downloaded earlier from the phone. I want to show you what smali code looks like so that you can get your hands dirty with it. If you have set up APKTool correctly, it should be in your path, so you can execute it from any directory that you're in. With that, on your terminal, change directories to the working one where you saved Lylt last time. From that directory, run `apktool d lylt.apk`. You can see what happens in the following code:

```
➜  lylt ls -alrt
total 19536
-rw-r--r--  1 sheran  staff  10000865 Mar 23 19:46 lylt.apk
drwxr-xr-x  6 sheran  staff       192 Mar 23 22:22 ..
drwxr-xr-x  3 sheran  staff        96 Mar 23 22:22 .
```

```
➜  lylt apktool d lylt.apk
I: Using Apktool 2.4.0 on lylt.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /Users/sheran/Library/apktool/
   framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Baksmaling classes2.dex...
I: Baksmaling classes3.dex...
I: Baksmaling classes4.dex...
I: Baksmaling classes5.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
➜  lylt
```

Now once APKTool is done, it would have created another directory called lylt in the current one. Change to that one and list the contents as shown as follows. In some cases, depending on app version, the file listing could be different.

```
➜  lylt cd lylt
➜  lylt ls -alrt
total 88
-rw-r--r--    1 sheran  staff  11743 Mar 23 22:23 AndroidManifest.xml
drwxr-xr-x  144 sheran  staff   4608 Mar 23 22:23 res
drwxr-xr-x    4 sheran  staff    128 Mar 23 22:23 smali_classes4
drwxr-xr-x    3 sheran  staff     96 Mar 23 22:23 smali_classes5
drwxr-xr-x   86 sheran  staff   2752 Mar 23 22:23 kotlin
-rw-r--r--    1 sheran  staff  12346 Mar 23 22:23 apktool.yml
drwxr-xr-x    5 sheran  staff    160 Mar 23 22:24 ..
drwxr-xr-x   10 sheran  staff    320 Mar 23 22:24 smali
drwxr-xr-x   14 sheran  staff    448 Mar 23 22:24 .
drwxr-xr-x    5 sheran  staff    160 Mar 23 22:26 original
drwxr-xr-x   29 sheran  staff    928 Mar 23 22:26 unknown
```

```
drwxr-xr-x    8 sheran  staff    256 Mar 23 22:26 smali_classes2
-rw-r--r--@   1 sheran  staff  14340 Mar 23 22:27 .DS_Store
drwxr-xr-x   12 sheran  staff    384 Mar 23 22:27 smali_classes3
➜  lylt
```

What you generally want to look for is in the AndroidManifest.xml and the other "smali_classes" directories. The Android Manifest of course contains essential information about the app. You will find a lot of information ranging from package name, what components exist in the app, compatibility, permissions, and where the start points of the apps themselves are. This would be the app entry point with which you can begin your tracing. The entry point is the one that gets executed when a user taps the app icon. To find this, you have to look for the Activity name that has the MAIN action type and the LAUNCHER category. If we look through Lylt's Manifest, we can see that `com.lylt.customer.feature.onboarding.OnBoardingActivity` is the entry point.

Now that we know this, let's find it from the terminal. If you use the Unix find command, you can easily find the file in question. See the following:

```
➜  lylt find . -type f |grep OnBoardingActivity
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentImpl$1.smali
./smali_classes2/com/lylt/customer/di/ActivityBuilderModule_BindOnBoarding
Activity$OnBoardingActivitySubcomponent$Builder.smali
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentImpl$OBM_BFAUD_ForceAppUpdateDialogSubcomponentBuilder.smali
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentImpl$2.smali
./smali_classes2/com/lylt/customer/di/ActivityBuilderModule_BindOnBoarding
Activity.smali
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentImpl$OBM_BFAUD_ForceAppUpdateDialogSubcomponentImpl.smali
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentImpl$OnBoardingFragmentSubcomponentBuilder.smali
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentImpl$OnBoardingFragmentSubcomponentImpl.smali
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentBuilder.smali
```

```
./smali_classes2/com/lylt/customer/di/ActivityBuilderModule_BindOnBoarding
Activity$OnBoardingActivitySubcomponent.smali
./smali_classes2/com/lylt/customer/di/DaggerAppComponent$OnBoardingActivity
SubcomponentImpl.smali
./smali_classes2/com/lylt/customer/feature/onboarding/OnBoardingActivity_
MembersInjector.smali
./smali_classes2/com/lylt/customer/feature/onboarding/OnBoardingActivity.smali
./smali_classes2/com/lylt/customer/feature/onboarding/
OnBoardingActivity$Companion.smali
➜  lylt
```

I use the command `find . -type f |grep OnBoardingActivity` which prints out every file name in this directory and does so recursively through every subdirectory but then only shows me the ones that have `OnBoardingActivity` in the name while being case sensitive. The one we're after is this one: `./smali_classes2/com/lylt/customer/feature/onboarding/OnBoardingActivity.smali`.

I use Sublime Text 3 as my text editor, and I love it. I paid for it, and it is fully worth it. Sublime can be extended through its built-in package manager. It also has syntax highlighting that you can download through this package manager, and I have downloaded and installed the smali syntax highlighter. I think it is invaluable. Otherwise, you will be staring at white text with little context if there's no highlighting. Figure 5-13 has an image of the file opened in Sublime Text 3 with smali syntax highlighting. Another good option to consider is Visual Studio Code as it is another tool that has a large number of extensions available for use.

```
# instance fields
.field private final mActivity:Lcom/redteamlife/aas2/aas2client/MainActivity;


# direct methods
.method public constructor <init>(Lcom/redteamlife/aas2/aas2client/MainActivity;)V
    .locals 1
    .param p1, "activity"    # Lcom/redteamlife/aas2/aas2client/MainActivity;

    const-string v0, "activity"

    invoke-static {p1, v0}, Lkotlin/jvm/internal/Intrinsics;->checkParameterIsNotNull(Ljava/lang/Object;Ljava/lang/String;)V

    .line 11
    invoke-direct {p0}, Landroid/os/AsyncTask;-><init>()V

    .line 13
    iput-object p1, p0, Lcom/redteamlife/aas2/aas2client/NetworkAsyncTask;->mActivity:Lcom/redteamlife/aas2/aas2client/MainActivity;

    return-void
.end method


# virtual methods
.method public bridge synthetic doInBackground([Ljava/lang/Object;)Ljava/lang/Object;
```

Figure 5-13. *Sublime Text with open smali file and smali syntax highlighting*

It is easy enough to start analyzing an app with APKTool and some Unix magic together with Sublime Text. You can still get the job done in terms of tracing program flow, figuring out which component does what, and so on. I think this would take a little longer though, because of a lack of a unified project management interface. I did an experiment with a Unix one-liner to generate a tree out of the directories we had generated. Had I used it in this book, even formatted for it to read like code, it would take up 20 pages in this chapter that I don't think it would be worth anyone's while in including – no matter how fat this book got.

Therefore, I must reluctantly bid goodbye to APKTool and its friends when I analyze apps. I want to reiterate that like a mechanic working on a high-performance car, in the absence of time to make them, the best thing to do would be to buy some high-quality tools. This is why I will now move on to JEB.

JEB

JEB, if I have not said before, is one of my favorite tools in my toolbag. Some may find it priced quite high, but I feel it is quite worth the payment – especially on the topic of getting good tools to get the job done. I consider JEB to be a self-contained reverse engineering toolkit. It has a user interface that puts a lot of very useful features at my fingertips and that I can view at a glance. Figure 5-14 shows the typical JEB interface. I will briefly describe the components here:

Main window: This is where you see disassemblies, bytecode, and decompiled source code. It also shows you a graphical view of the various code components so you can easily visualize branching and jumps.

Explorer: This is the project explorer. You can see and open files such as the Android Manifest, certificates, and resources such as strings and image.

Hierarchy: This window shows the directory tree of the disassembled apk. The source files that were packaged into this apk will be visible for you to navigate. This includes third-party libraries and the app's code.

Logger: This window shows you JEB's execution and what it is doing in the background. Errors, warnings, and others will show up here. You also get access to a terminal that you can use while debugging.

Callgraph: From here, you can generate the callgraph for your apk. Essentially, a callgraph is a visual representation of all calls made by your app to other parts of your app, third-party libraries, or frameworks.

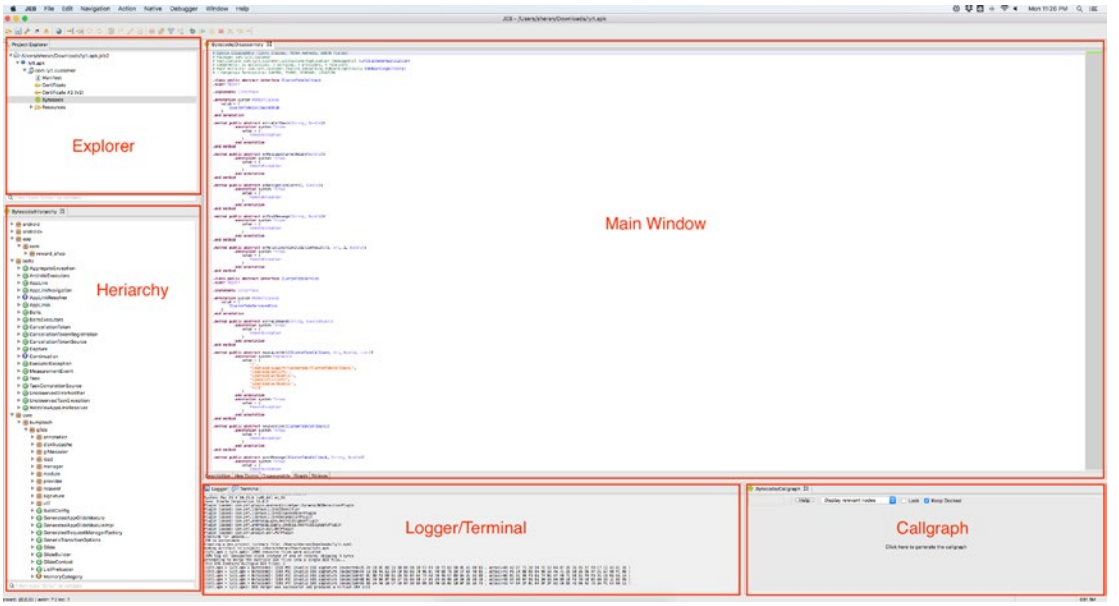


Figure 5-14. *JEB and how I have my version setup*

For the longest time, I used to wonder about the name JEB, until I asked Nicholas, the founder and creator of the app. I will let him tell you in his own words when I asked him:

> *Hello Sheran – yes, not a problem. I was struggling to find a name at the time (early 2013), and since the original version was focused solely on reversing Android apps, I thought 'okay, this complements IDA, in a way it's IDA+1,' and thus, by shifting all 3 letters by one position, we end up on JEB. I liked it and it started to stick, so I kept it.*

The actual process of conducting a static analysis varies greatly, but what is common is to understand what source code triggers a certain behavior in an app. One common use case of static analysis is to discover how data is encrypted within an application. If someone has rolled their own encryption or encoding mechanisms, we can easily uncover it in a session of static analysis. Another reason would be to see how or whether sensitive data was stored on a device – data such as an encryption key, a private key, or a shared secret that you can later use to decipher communications between the app and the server. Other things I look for when doing static analysis are source code that may generate or validate a specific code. For example, if the app requires you to enter a serial number or specific key code, then I take great joy in discovering how to defeat that check for a valid key entry. I will spend days if I have to tracking down the piece of code where the check is done to verify whether a specific code entered is correct. Usually, this is the gatekeeping function that will see if you have the correct code in order to unlock a specific feature. Two ways to go about defeating this would be (1) to learn how the algorithm works and write code using that algorithm to generate your own keys or codes – the older of you would know this as keygenning – and (2) to bypass the check altogether. Usually, the gatekeeper has at its core a very basic if-else function. If key is correct, allow access; if not, deny. So, one thing you can do is patch the apk or source code and rebuild the app so that your version allows access even if the code is incorrect.

If you don't have JEB and are wondering what your options are, then there are a few. APKTool is one which will decompile your APKs into smali code, which is still easy enough to understand, but obviously not as great as having Java code. A really good tool combo to use is dex2jar and JD-GUI, which we covered briefly in the previous chapter. Essentially, it would involve converting your APK into a JAR file and then decompiling the JAR file through JD-GUI. You don't get the cool debugger or the additional on-the-fly decryption of obfuscated strings and such, but it can still be pretty effective. In place of a debugger, you can consider using Frida which does runtime instrumentation. That means that you can locate and pause and alter bits of the app while it is running. Frida does require rooted devices for it to run well, however.

I'd like to highlight a few more screens from JEB and its features. Figure 5-15 shows the decompilation of the Lylt app's OnBoardingActivity. Then Figure 5-16 shows the source code hierarchy and how easy it is to navigate. Lastly, Figure 5-17 shows the graphical representation of the code to visualize the program flow.

```
import kotlin.Metadata;
import kotlin.jvm.internal.DefaultConstructorMarker;
import kotlin.jvm.internal.Intrinsics;

@Metadata(bv = {1, 0, 3}, d1 = {"\u0000*\n\u0002\u0018\u0002\n\u0002\u0018\u0002\n'
public final class OnBoardingActivity extends DaggerAppCompatActivity {
    @Metadata(bv = {1, 0, 3}, d1 = {"\u0000\u0012\n\u0002\u0018\u0002\n\u0002\u001(
    public static final class Companion {
        private Companion() {
        }

        public Companion(DefaultConstructorMarker $constructor_marker) {
        }
    }

    public static final Companion Companion = null;
    public static final String KEY_REQUEST_ADDING_SHOP_UUID = "uuid";
    private HashMap _$_findViewCache;
    @Inject
    public Navigator navigator;
    private String uuid;

    static {
        OnBoardingActivity.Companion = new Companion(null);
    }

    public void _$_clearFindViewByIdCache() {
        HashMap v0 = this._$_findViewCache;
        if(v0 != null) {
            v0.clear();
        }
    }

    public View _$_findCachedViewById(int arg3) {
        if(this._$_findViewCache == null) {
            this._$_findViewCache = new HashMap();
        }

        View v0 = (View)this._$_findViewCache.get(Integer.valueOf(arg3));
        if(v0 == null) {
            v0 = this.findViewById(arg3);
            this._$_findViewCache.put(Integer.valueOf(arg3), v0);
        }

        return v0;
```

Figure 5-15. *JEB's decompilation of the Lylt MainActivity*

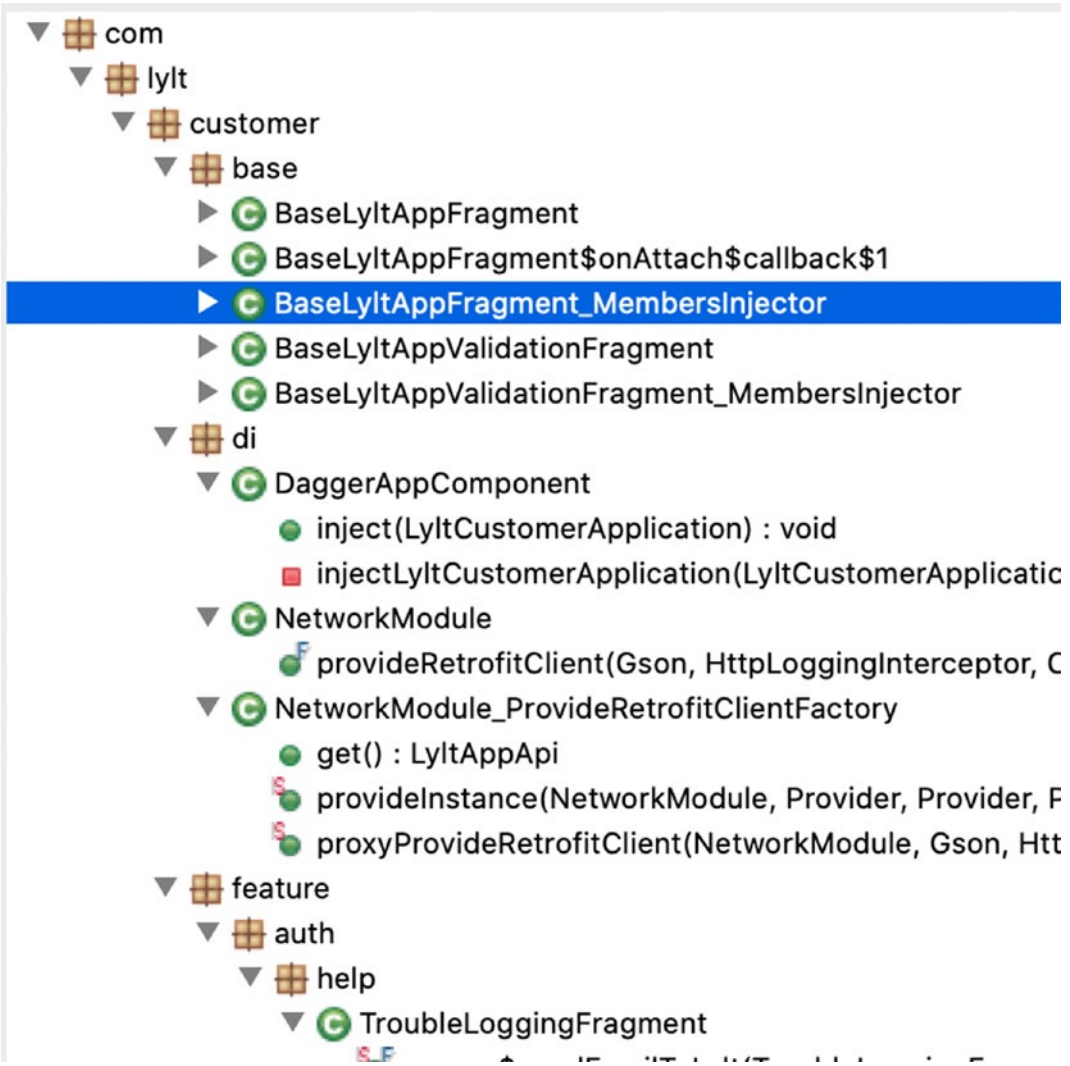


Figure 5-16. *JEB's source code hierarchy*

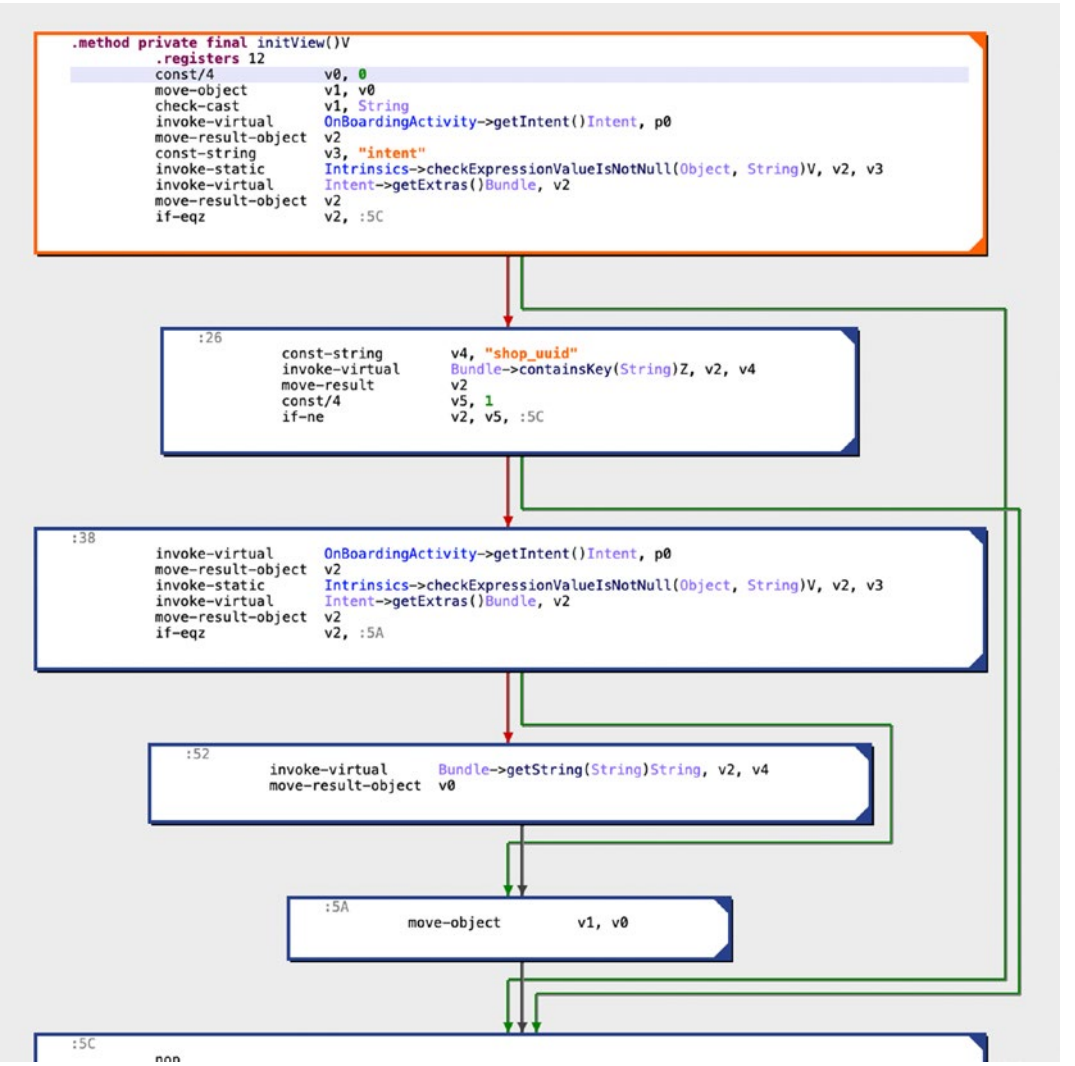


Figure 5-17. *JEB's visual program flow representation*

As an attacker, and a helpful one at that, what your aim should be is to find instances where the engineering team or developer has made less secure decisions when writing their code. These instances can be used as tools to further their learning. I find that taking the time to learn and understand the context of the code and then presenting the actual code to the developer, sitting with him, and working through the reasons are more powerful than taking the approach of "Oi I'm security, do what I tell you to do." Of course, you may have to invest some time and energy in learning to communicate with less receptive people. This is because from the beginning, the sitting down of developer

and security engineer is viewed as adversarial. It never is. The adversary is out there – the hacker that is trying to wreak havoc on your systems, not the guy working in the same company trying to figure out together with you how to solve a particular security concern. In practice, this is rarely the case. The us vs. them mentality is prevalent throughout startups and organizations. I was personally responsible for propagating such a message to my team, not explicitly but by cheering them on and acknowledging how good they were. My praise of them began to take on a note of pride in competence. We were so good that we could take apart any developer's code. Praise is important; hubris is unforgivable. Learn to work well with your engineering team. It will pay dividends in the long run.

CHAPTER 6

The Tool Bag

The techniques in this book are heavily reliant on external tools. This book does not teach you how to develop those tools but attempts to enlighten you on how you can make use of these tools to both build and test your apps. That's why I have dramatically decided to call this chapter "The Tool Bag." I will also spend a little time telling you about my setup that I use when reverse engineering and debugging Android apps. Your mileage will considerably vary, and perhaps you do not want to or are unable to set up your environment this way. This is fine. You can still get the job done, so don't fret too much. In this chapter, I will talk about tools that I use to build, test, and break Android apps. The list will be a combination of free and commercial tools that I use personally in my daily routines depending on whether I am wearing a builder hat or breaker hat. I skew toward breaker by nature and by the career that I have had. This may be apparent in the dearth of the builder tools covered here, but it will cover the basics of what you need so that you can build your apps and then ensure that they are secure.

On the topic of breaking, I'd like to take this time to discuss effectiveness. You can only be as effective as your tools are, unless you are an author of your own tools. Yes, you can make some headway with that, but in my case, I have neither the interest nor do I have the patience to build my own tools. Having said that, there are tools and then there are tools. In some cases, you can get a free tool that does about 60% of what you want, and the rest you can plug by writing some of your own scripts. You may find a multitude of such fragmented tools which you can incorporate into your workflow with your own "glue" scripts that hold things together. I find this takes a bit longer and requires significant experimentation and research to become useful on a project basis. What do I mean by this? Well, if your nine-to-five was solely taking Android apps apart and checking their security, then the fragmented approach I described earlier works better. However, if you did other things in addition to breaking Android apps like I do, then I find it more effective to invest in purchasing tools. The commercial tools will help you achieve a greater deal and will also mean you switch less between each tool and your custom scripts, and thus your workflow will be smoother. This will become clearer as we explore the tools.

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_6

The Builder Tools

Let's start with some of the tools that I have and still use to build Android apps. The focus for this section is to outline a set of tools that not only you will use to write actual code but will also include tools that help you quickly debug and review your own source code for bugs or errors.

Android Studio

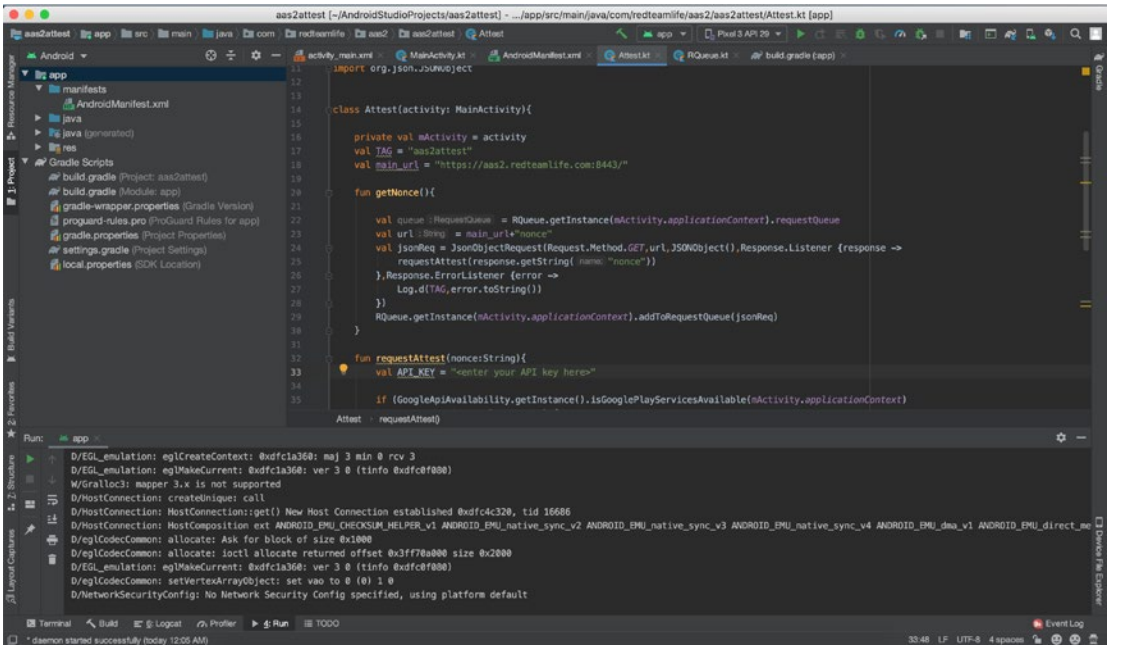


Figure 6-1. *A screenshot of my Android Studio with opened sample project on MacOS*

One-stop shop comes to mind when considering Android Studio. Launched in 2014, it was an IDE that was designed and built specifically for Android development. The platform uses the IntelliJ IDEA software base that was built by JetBrains. It contains a suite of features to help Android developers get started on their apps quickly and is a firm favorite of mine to use. Versions exist for MacOS, Linux, Windows, and even Chrome OS if you want to do some app development on Google Chromebooks. I use MacOS, so my discussions will be centered around that.

I used to use Android Studio for my Android app development. Now I mostly use it for the AVD, which is the Android Virtual Device Manager, and for writing up snippets of

code to compile and then eventually decompile so that I can become familiar with how specific types of Android source code looks like in bytecode when reverse engineered. The AVD is a really useful tool to me because it allows the creation and control of multiple emulated Android devices. I like it because it allows me to rapidly test apps or configurations on emulated devices without the need of having a specific hardware testing device. It is also great for continuity because you can have multiple emulators saved in your AVD in a paused state. When you work on an emulator and then pause it, the state is saved so you can switch to a different task and then return to the emulator and pick up where you left off. My current AVD looks like the one shown in Figure 6-2. You can invoke it from the `Tools` ➤ `AVD Manager` menu on MacOS when you have a project open; otherwise, use the button `Configure` ➤ `AVD Manager` on the Welcome screen.

Figure 6-2. *The Android Studio AVD*

Each virtual device that you create will need you to tell it a specific Android version to run. Android Studio's SDKs date as early as Android 2.1 (Eclair) API level 7. If you want to geek out with the numbers, you can find a list of codenames, tags, and build numbers on the Android source website at this URL: `https://source.android.com/setup/start/build-numbers`.

Having older SDKs is really great not just from a developer standpoint but also from a debugging or security standpoint. As a developer, you will be aware that the rate of adoption of newer Android versions is painfully slow. So those cool new features Google released can't always be used in your app until a lot more people adopt the

new version. As a hacker, that usually works in your favor because older versions may have vulnerabilities, and a user's reluctance to upgrade is ideal for your preying hands to take advantage of. Google releases a distribution dashboard that shows the percentage distribution of each Android version here: https://developer.android.com/about/dashboards. The data on that page was up to 7 May 2019 and showed that Android version 9 (Pie) had a 10.4% distribution. As a comparison, Android Pie was released in August of 2018. Thus, after a ten-month period from release, there was a 10.4% adoption rate.

My Android Studio Tweaks

With my Android Studio, I don't do much extra. I will first make sure I have the latest Android version downloaded so that I can create a virtual device using that SDK. If I need older SDKs, then I will use the SDK Manager (Figure 6-3) and download the older version.

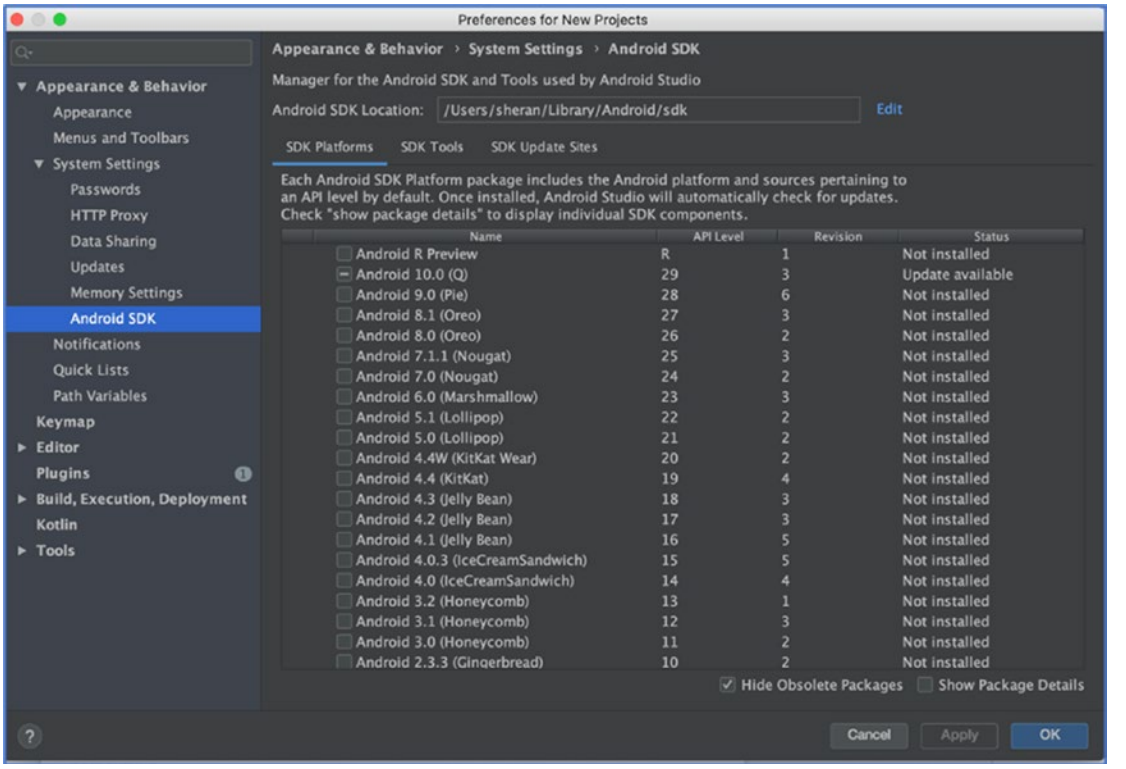


Figure 6-3. *Android Studio SDK Manager*

Access the SDK Manager by using either `Tools` ➤ `SDK Manager` if already in an open project or the button `Configure` ➤ `SDK Manager` if on the Android Studio Welcome screen (Figure 6-4).

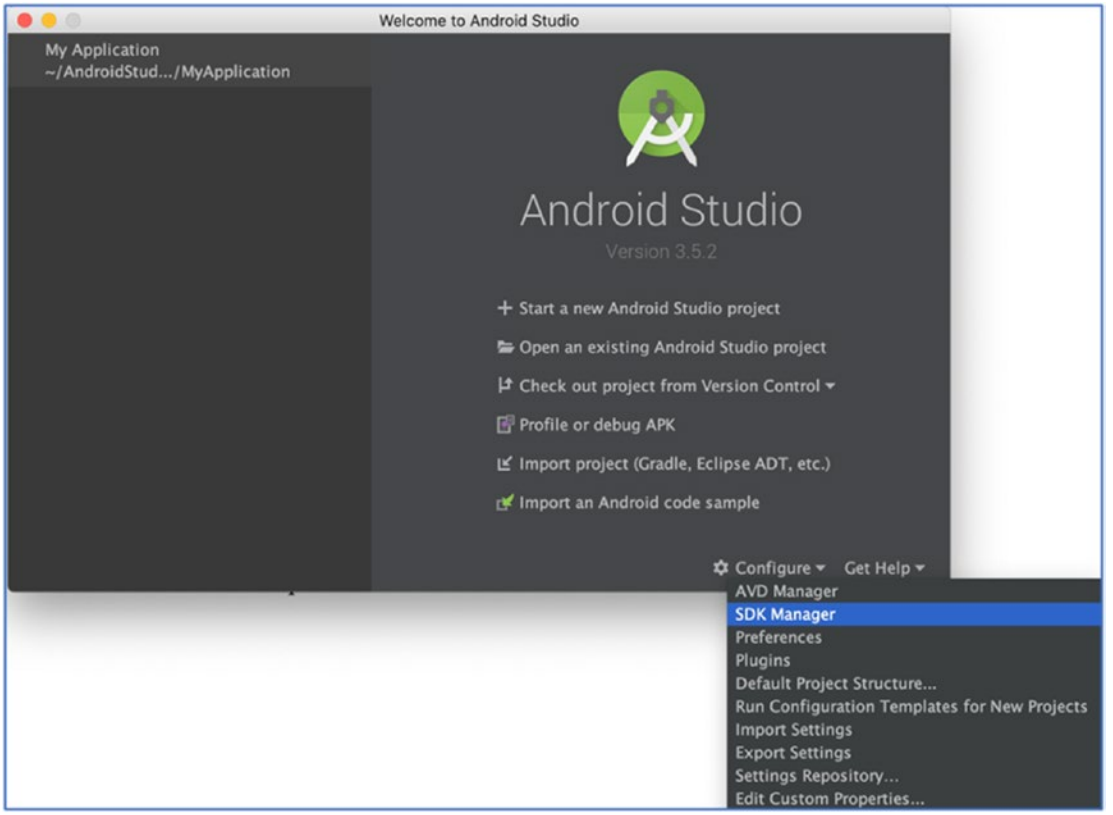


Figure 6-4. *Accessing either the AVD or SDK Manager on Android Studio*

Creating a Virtual Device

Let's go through the steps of creating a virtual device so that I can address one topic – the Google Play Store. Let's fire up the SDK Manager first and get us a new SDK. If you're on the Welcome screen, then hit the Configure button and select the SDK Manager (Figure 6-4). This should then bring up the SDK Manager that looks like Figure 6-3. Next, we select an SDK of our choosing. For this example, I will choose SDK version 9 (Pie), API level 28. Click the checkbox in the leftmost column of the SDK Manager in line with the SDK version of your choosing. You will see a little download icon highlighted next to it as in Figure 6-5.

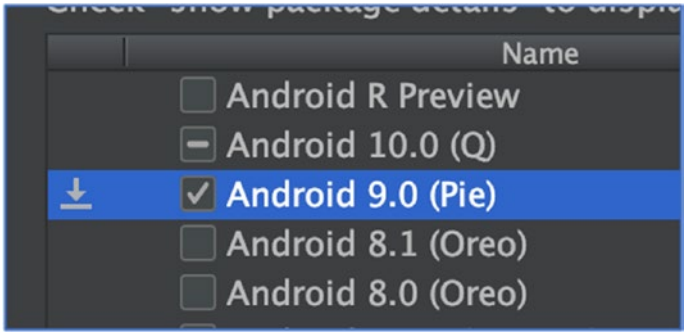


Figure 6-5. *Selecting the SDK to download in Android Studio*

When you're done, click the OK button, and Android Studio will prompt you to confirm your request. It will also let you know how big the download is and how much space it will take up on your disk so you can be prepared for that as well (Figure 6-6).

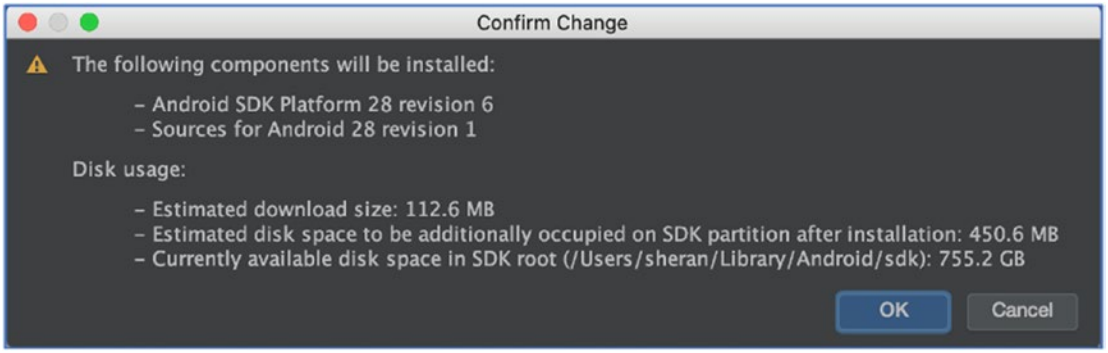


Figure 6-6. *Confirmation prior to download of the SDK*

You will see a progress indicator window where you can track the progress of the download and install of the SDK. Any errors taking place will also be visible there. Once the download and installation complete successfully, you can dismiss the window by clicking the Finish button. If all went according to plan, you will have the Android version 9 SDK downloaded and installed on your Mac. The next step is to open up the AVD Manager and create a new virtual device with this SDK. Let's go back to the Android Studio Welcome screen and click the Configure button and this time choose AVD Manager. This brings up the AVD window where you will see a list of previously created virtual devices. If this is the first time you installed Android Studio, then you won't see any virtual devices. Click the `+ Create Virtual Device` button and a window will pop up asking you to select your hardware as in Figure 6-7.

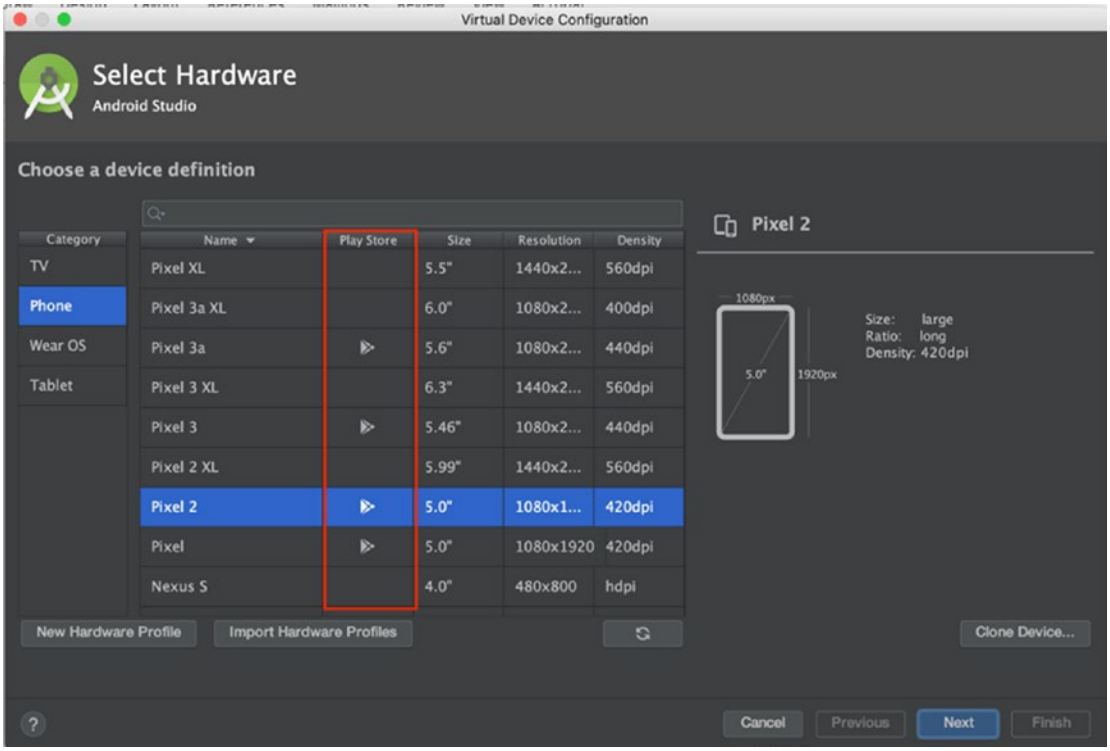


Figure 6-7. *Hardware selection when creating a new virtual device*

You will notice on the second column named "Play Store" some devices have a Google Play Store icon. These devices are special in that they have the Google Play Store and all other accompanying Google services, collectively known as Google Play Services, installed. This means that you can use your Gmail account or any other Google registered account to log in and configure that device just like you would a physical device. What this also means is that the virtual device will not be rooted. Therefore, you cannot use the Android Debug Bridge (ADB) to execute programs as the root user, nor can you freely look through the Android filesystem through ADB. Google says the following on their site:

To ensure app security and a consistent experience with physical devices, system images with the Google Play Store included are signed with a release key, which means that you cannot get elevated privileges (root) with these images. If you require elevated privileges (root) to aid with your app troubleshooting, you can use the Android Open Source Project (AOSP) system images that do not include Google apps or services.

The AOSP images that they refer to are the ones without the Google Play icon. They do not contain any of the Google Play Services. This isn't a hindrance in any way; if I need to run an app on a rooted emulator, I will install the AOSP image instead. In Figure 6-7, you will notice that Pixel 2 is a system image that has Google Play Store installed which will be non-rooted. If you want a similar device as that but with root, install the Pixel 2 XL image.

Let's finish up installing this virtual device. Select the Pixel 2 XL image and click the Next button. Then you get to select the Android version that you want. This should be the SDK image that you downloaded earlier, so select Pie. Sometimes, you will also need to download a system image and accept the end-user license agreement. If the Release Name column has a "Download" link there, click the link, then read and accept the license agreement on the next window, and then click the Next button as shown in Figure 6-8.

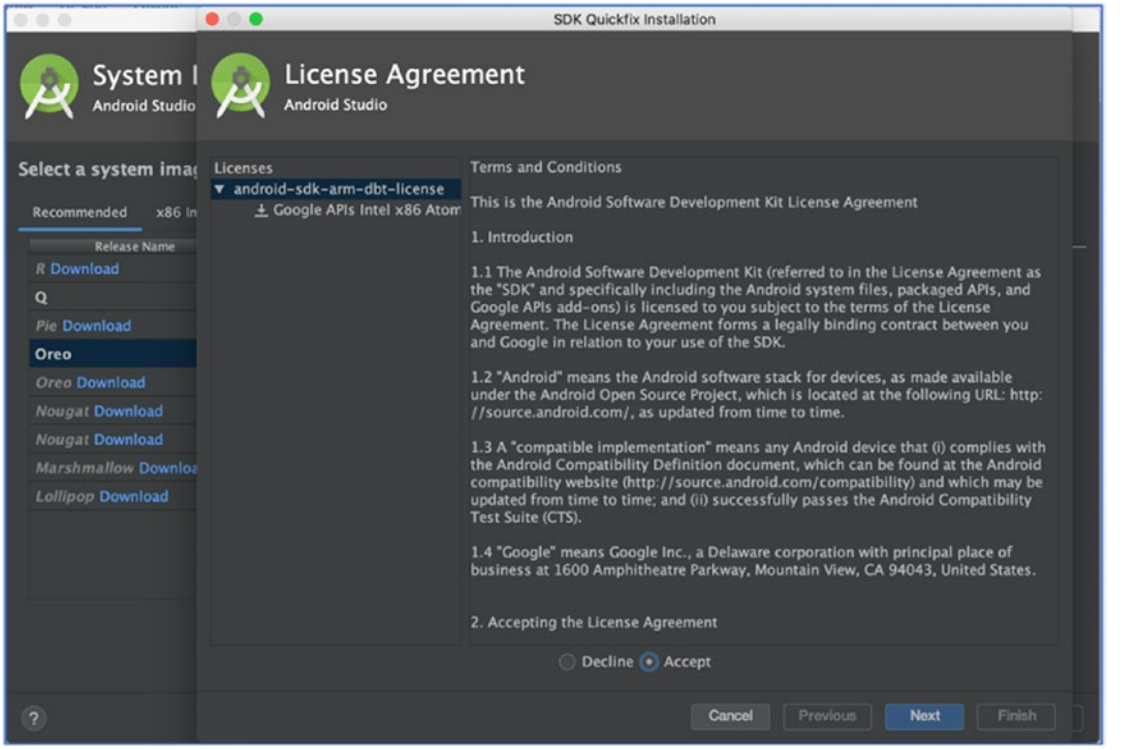


Figure 6-8. *Reading and accepting the license agreement to download the system image*

When the download is done, click the Finish button, then select the Android Pie system image, and click the Next button. You will then see a window that looks like Figure 6-9. From here, you can select things like the device orientation when it starts up (whether landscape or portrait), what type of graphics, and whether to show a frame around the emulator that looks like an Android device. There are also several advanced options that allow you to select the number of cores on the CPU, memory and storage sizes, where the SD card resides, and the network type and speed (which is useful for simulating 3G networks at varying speeds). For now, leave the settings as they are and click the Finish button.

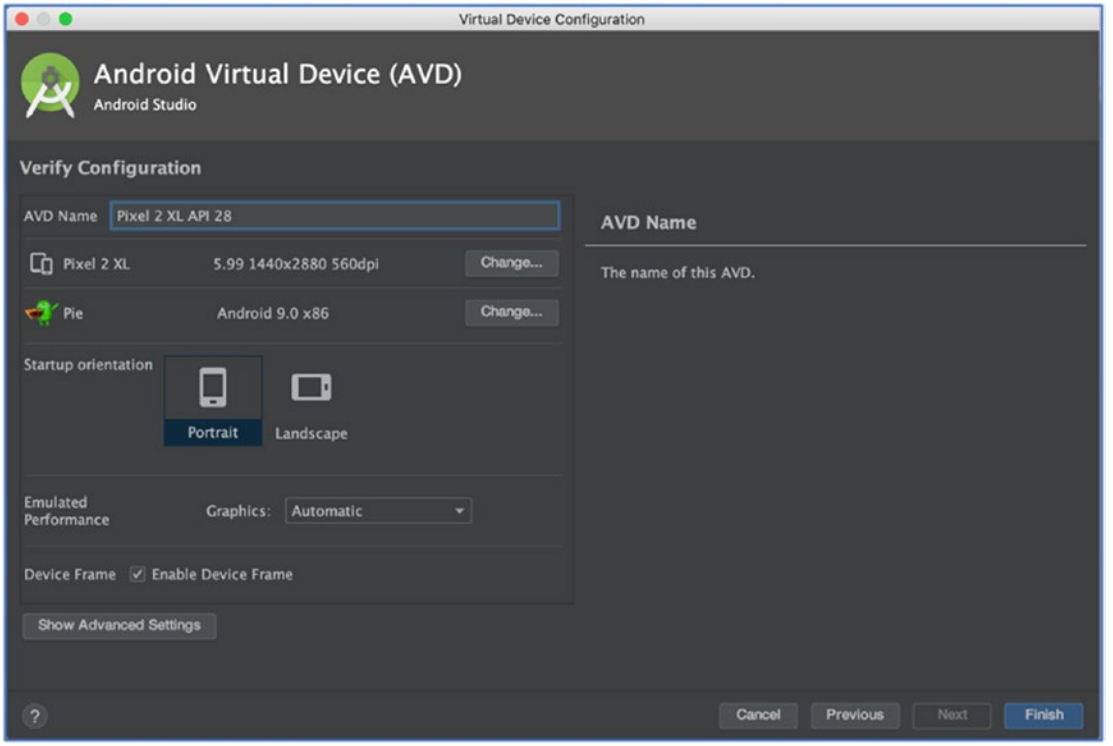


Figure 6-9. *Finalizing the virtual device you just created*

You will then be returned to the list of your virtual devices (Figure 6-2), and you will see your newly created AVD listed there. We won't fire it up now, but there will be plenty of instances when we will.

As far as emulators go, there are a few others out there including Genymotion and BlueStacks that offer differing solutions, especially if you want to run the virtual devices in a large group. For the purposes that I need, I find that the Android Studio emulator is a good fit.

The Breaker Tools

I'm not entirely certain, but I believe a fair number of readers were keener to get to the breaker without or before even considering the builder tools. I will outline some of my favorite and most useful breaker tools here.

Burp Suite – Web Application Security Test Kit

Figure 6-10. *A screenshot of my Burp Suite startup screen*

What can I say about my oldest friend Burp Suite (https://portswigger.net/burp)? I was using it when all it was was a proxy and the Windows app icon was a badly drawn face of a guy that looked like he was burping. Good times. Burp Suite today can easily be described as a testing framework for web application servers. At its core lies a very good crawling engine that helps gather all possible URLs in a web application and its man-in-the-middle (MiTM) proxy. I use Burp to test web applications mostly, and given my

familiarity with it, I use it when I want to test what Android apps send back and forth to their back-end servers. I do this primarily by using the Burp Proxy, and then I can use the HTTP inspector to see real-time HTTP requests going back and forth between the app and the server as shown in Figure 6-11. This is one tool that I can recommend that you purchase. The Professional version offers a lot more features that allow for a very seamless testing session and also includes more tools like the Burp Scanner which does automated scanning for web applications. Having said this, since we only need the proxy functionality, you should be able to get by with the Community version. But I would say, do yourself a favor and buy the Professional version.

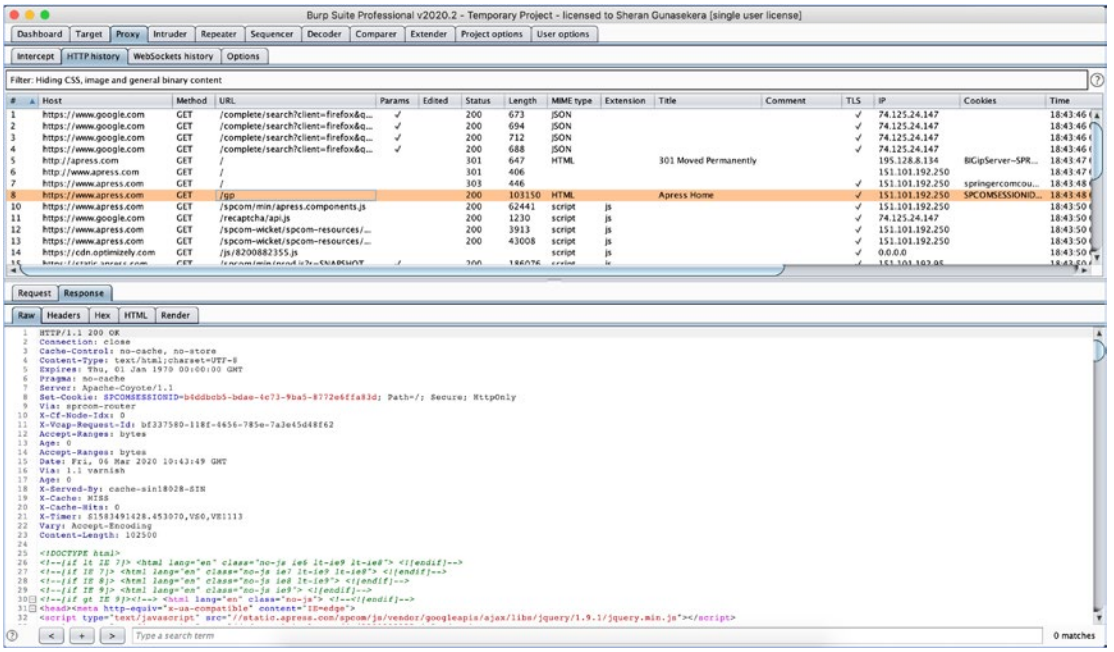


Figure 6-11. *Burp Proxy showing all the captured traffic between my Mac and the apress.com website*

One question that you may have is "Can I see TLS traffic?" and the answer is yes and no. Let's quickly revisit what an MiTM attack is. Figure 6-12 shows what it looks like when network traffic is directed to its intended server with one hop in between. This hop is called the man in the middle. Essentially, the job of the MiTM device or server is to receive any data that arrives via its server port and pass it on to the intended destination. It does this for every single network packet. You could say that all traffic going back and forth between the client and the server passes via this proxy.

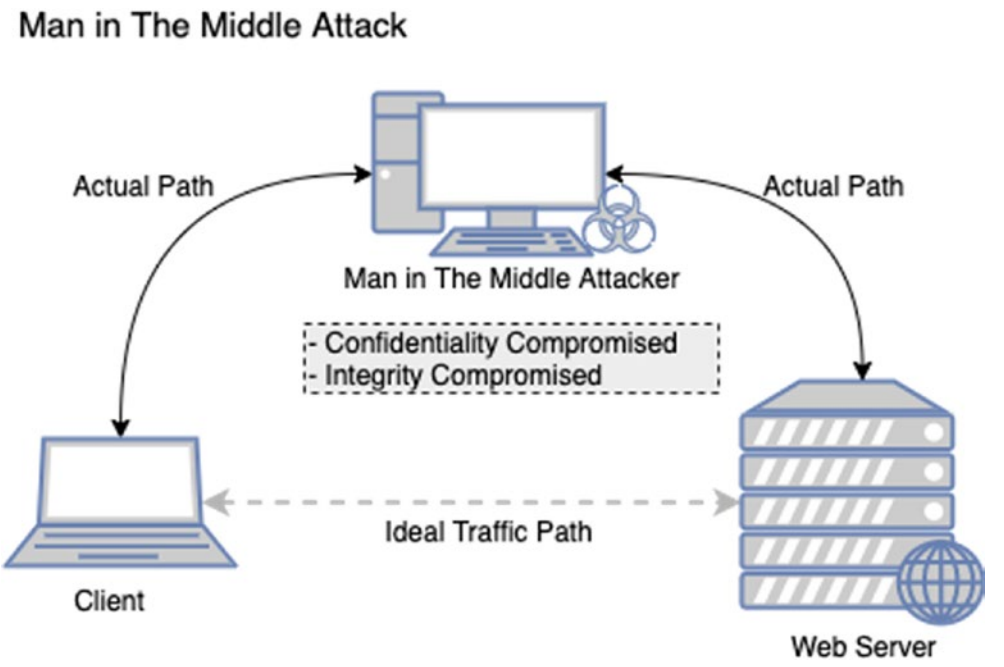


Figure 6-12. *Man-in-the-middle attack data flow*

That's step one. When you know that all the network traffic passes through this proxy, the owner of the proxy (hopefully you) can take a look at all this traffic that is going back and forth. This means that the data traveling between the client and the server isn't private. This is especially true if the client and the server are not part of your infrastructure or organization and belonged to a third party. You would be able to view all messages going between the two talkers which means you have compromised the confidentiality of that data. Now if you chose to, you could also compromise the integrity of those messages. Let's say the client sends a chat message from one user to another via the server. You could watch for some keywords of this chat message, and when you intercept these keywords, you can change the text and then pass it along to the server. You can do this because you are receiving the message and you are also forwarding it on behalf of the party you receive the message from. Now you have compromised the integrity of that message as well. If you look back over your old information security notes on the CIA Triad, no, not the CIA with the Triads, the CIA Triad. The triangle with the three words forming the acronym CIA: Confidentiality, Integrity, and Availability. You will see that through this attack, we have managed to compromise two of the three concepts. Of course, presently, information security is far more nuanced and complex, but if we simplify into the triad, we can see that the MiTM attack is quite an effective one.

Yes, yes, of course, I hear all of you at the back of the class saying that TLS will prevent the compromise of both of those concepts. That is, after all, why you're all here, right? This is the part where we answer "yes" to the question "Can an MiTM attack see TLS traffic?" It takes a little bit of prep work beforehand. Broken down, it goes something like this:

1. Install our own Certificate Authority (CA) certificate on the Client
2. Mark that CA certificate as trusted on the client side
3. Generate multiple TLS certificates for each host that the client visits which are signed by our CA

Note If we wanted to, we could build a proxy that did the three steps outlined here. This would take time to build and test. Tools like Burp Suite have this functionality built-in and is easily usable. This goes to my point about being effective with the tools that you use.

Why does this work? Ultimately, it is because we have installed our CA certificate and marked it as trusted on the client. When we do this, the client will trust any certificate issued under that CA implicitly. Therefore, with this setup, we are able to see even encrypted TLS traffic that passes between the client and the server and, yes, even edit the data.

Let's address the "no" part of this answer to the same question "Can I see TLS Traffic?" I said no because of the prerequisites that we need to fulfill prior to getting a successful TLS traffic dump between the client and the server. We would have to successfully convince the client to send all his traffic to us so that we can forward it along to the intended recipient. That isn't hard by itself. We could easily do that if we were sitting on the same network as our client. We can run an ARP Spoofing attack on the client and trick him into believing that we are his next hop gateway rather than just another client. Today, however, there are many ways to prevent ARP Spoofing attacks, and to make things more complex, an ARP Spoofing attack, if not done correctly, can have some catastrophic results.

Speaking of ARP Spoofing, I remember a long time ago, an ex-colleague of mine once forgot that he had an active ARP Spoofing attack running on his laptop during an assessment where essentially about 30 servers on the same subnet were sending his modestly powered device all their traffic. Surprisingly, his laptop was able to cope with the data, and everything went well. At least until it was time for him to head to a

meeting to update the customer on the progress of our security assessment thus far. Having forgotten the ongoing attack, he swiftly closed the lid of his laptop, unplugged the Ethernet cable, and proceeded to head to the meeting that never took place. About 2 minutes after he removed his laptop from the network, alarms began to go off, and we saw nervous and confused looks all around as customer employees began shouting and running around. Basically, the result of unplugging the laptop from the network of servers that all thought you were the gateway or the router meant that all of the servers were sending traffic to a nonexistent device. The attack had overwritten the ARP address of the actual router, and so, it only knew this one hardware address to send data to: my colleague's laptop which was no longer there. No traffic was sent or received correctly on that production network subnet for sufficient time that the customer's major services went straight down and stopped responding. We had to immediately plug his laptop back in and re-ARP or tell all the servers that the real router was at the correct hardware address. A restart of all servers would have also worked, but we were in no way willing to trust that the restart of 30 servers would go off without a hitch. So we had to re-ARP which we did, and gradually everyone began to relax as the services all came back up again. So, be extra careful if you want to try ARP Spoofing these days, but I am fairly certain that the ARP Spoofing attack is a thing of the past.

The next challenge would be to get the user of the client PC to install and trust our CA certificate. The best way to do this would be when he leaves his PC unlocked and walks away to go get a snack or visit the toilet. Otherwise, you're looking at complex hacking attempts on his PC to mimic how he would download, install, and trust the CA certificate, and this would take considerable time and carry lower probabilities that it would succeed. With these kinds of odds, you can see why I would bring in the "no" answer to that question. I guess the real answer is "It depends" – in the context that in this scenario, we have little to no control over the client and what it is doing. The only real way we could succeed at breaking TLS traffic is *if we were to fully and completely own the client*. Humor me and follow this train of thought on context. What scenario gives us complete control over the client device where we could introduce and trust our own CA certificate on the client device? Well the answer is simple: It's on our mobile phones that run third-party Android apps. We have complete and total control over our mobile phone. We can install the fake CA certificate, mark it as trusted, and then route all the traffic from a third-party app through our proxy to the ultimate destination and back.

Attacks like this are the reason why SSL Pinning was created. SSL Pinning is a technique where certain specific and trusted certificates from the app creator are embedded within the app. The app will run a comparison on whether the certificate

presented to it when connecting to a server matches the one embedded within the app. If it does not, then the connection is terminated so that no traffic flows between the client and the server. No data means anyone watching will be sorely disappointed. But as you all may or may not know, SSL Pinning has also been defeated, and the exact mechanism of breaking SSL Pinning will be discussed in several chapters later in the book.

My Burp Suite Tweaks

There's nothing noteworthy on this for the moment. I can really get good work out of Burp Suite's Professional version with minimal configuration. Both the free and commercial versions of Burp are well tuned to a pentester's perspective. Therefore, you will find yourself well setup from the beginning.

Frida – Dynamic Instrumentation Toolkit

Frida (`https://frida.re/`) is an awe-inspiring toolkit. Written by Ole André Vadla Ravnås, it is one of the few truly complete and extensible toolkits that allow both developers and security researchers access to some very powerful features. I personally feel that it is generally left alone by people that do not favor working on the command line. Frida is all command line, all the time. Look at Figure 6-13. I did my best to conceptualize what Frida does. Now, Frida can run on a multitude of devices and provide you with direct access to view and modify processes running on that device. For the purposes of this book, we will only cover the Android capabilities of Frida.

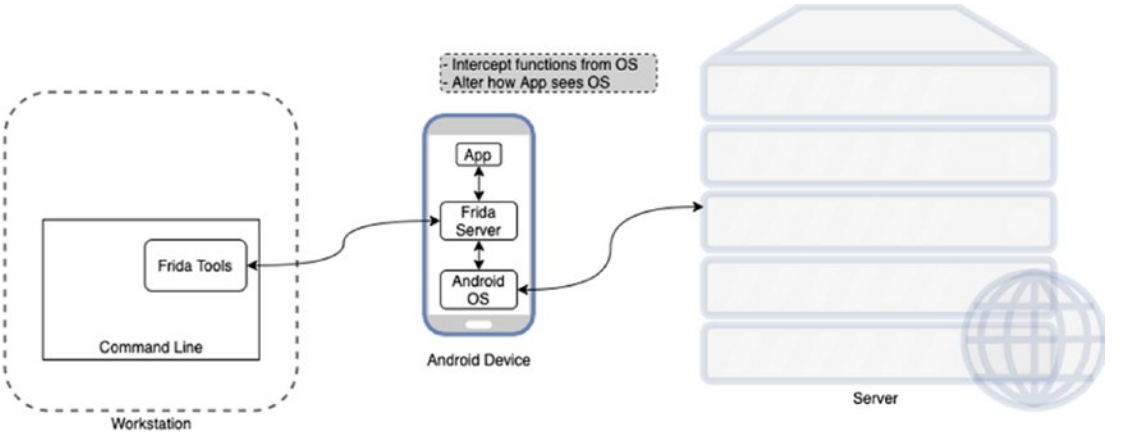


Figure 6-13. *An overview of how Frida works*

Frida has two components. First is the server component that runs on the Android device and hooks into the Android OS directly. Then it has the command-line tools component or client. The Frida client will send commands to the Frida server which can then oblige us by sending back instrumentation data, allow us to hook into different library calls or functions in the OS, trace private application code, or even allow us to replace and inject our own code into running apps. This sounds like a dream, and to be honest, you will feel that way when you can rip apart any Android app with ease and look into its innards – which APIs it talks to, what different servers it communicates with, what are the actual HTTP parameters being used – and look at what goes on behind the encrypted layer of TLS even if SSL Pinning has been implemented. I think many in the information security profession owe Ole a debt of gratitude for the tremendous tool that he has built and released for free. Thank you Ole!

Let's go a little bit deeper than just the surface here for Frida. I want to show you how simple it is to get going with Frida. In the later chapters on breaking SSL Pinning, we will be getting very detailed with how we use Frida, but for now just a taste.

INSTALLING FRIDA ON MACOS

OS X comes bundled with Python 2.7, and for this exercise, I will use that. I recommend you switch to Python 3. It is good. Use it. We will cover how to install it and get up and running with Frida on Python 3 later. For this example, we will use Python 2.7.

First, fire up your OS X Terminal.

1. Open Finder.
2. Click Applications in the left pane.
3. Scroll down in the right pane and open Utilities.
4. Scroll down in the right pane and open Terminal.

You should then see your Terminal window as in Figure 6-14.

Figure 6-14. *Screenshot of my MacOS X Terminal checking which Python version I have*

Let's now follow the steps you can see typed into the Terminal window.

1. Check and make sure you are in your home directory by entering `pwd`. You should see `/Users/<name you are currently logged on as>`. In my case, it was sheran. If you are not in the correct directory, you can type in `cd /Users/<name you are currently logged on as>`. Therefore, if you were logged in as sheran, you would type `cd /Users/sheran`.
2. Next, create a test directory that we will use to install our Python virtual environment and any files we want to keep there. Do this by issuing the `mkdir frida_test` command and pressing Enter.
3. Then, change to this directory by issuing the `cd frida_test` command and pressing Enter.
4. Lastly, we check to see what version of Python we have installed. Issue the command python –version and press Enter. If you receive anything that looks like Python 2.x.yy, then you have Python 2 installed.
5. Let's now install our Python virtual environment so that we don't affect the entire base Python installation. It is generally good practice to run your Python programs in its own virtual environment where you can independently manage requirements on a per project basis. To install the Python virtual environment with Python 2, type in `virtualenv -p /usr/bin/python2 python`. This should execute as shown in Figure 6-15 and return a new directory in your current one called python. In some cases, you may find that you don't have virtualenv installed on your system. If this is the case, then you can install it with: `pip install virtualenv`.

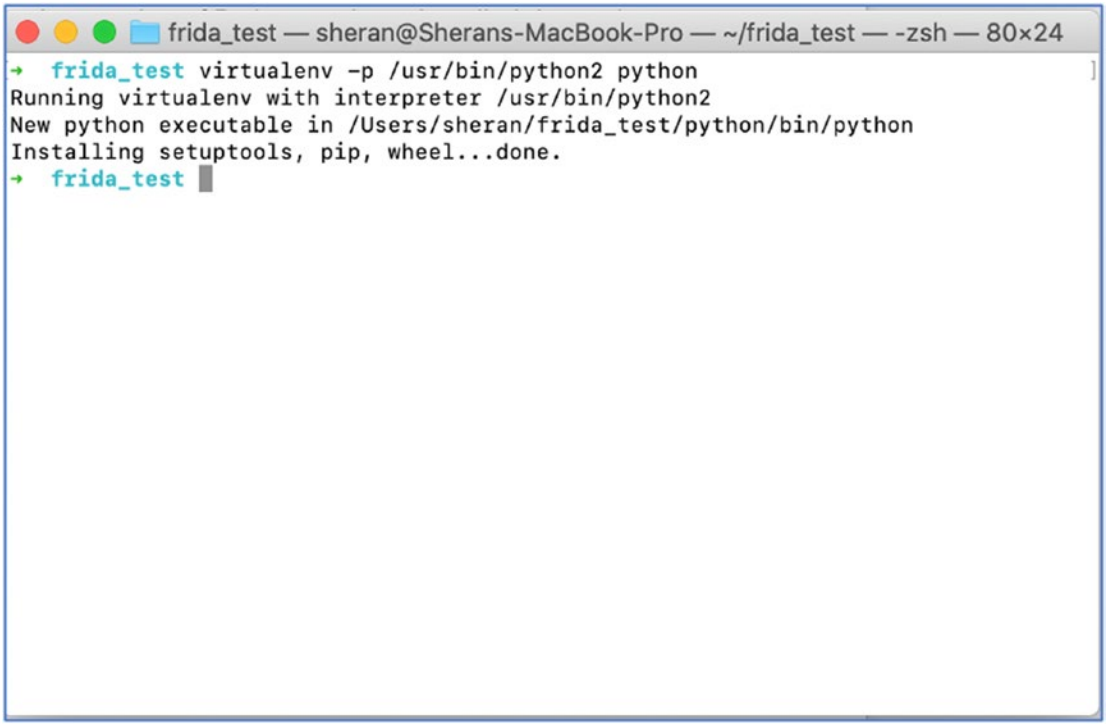


Figure 6-15. *Installing the Python 2 virtual environment*

6. Now we have to activate our new Python 2 environment so that we can install Frida and any relevant requirements. Type in `. ./python/bin/activate` to activate the virtualenv. Pay close attention to the fact that there are two period characters. If this looks confusing, then as an alternative use `source /python/bin/activate`.

7. Then let's install Frida by typing in `pip install frida-tools`. That's it!

```
(p3) → frida frida-ps -U
  PID  Name
-----  ------------------------------------------------
 6667  ATFWD-daemon
19748  abb
19576  adbd
  824  adsprpcd
  498  android.hardware.atrace@1.0-service
29288  android.hardware.audio@2.0-service
  851  android.hardware.biometrics.fingerprint@2.1-service
  663  android.hardware.bluetooth@1.0-service-qti
  499  android.hardware.boot@1.0-service
  664  android.hardware.camera.provider@2.4-service
  665  android.hardware.cas@1.1-service
  505  android.hardware.configstore@1.1-service
  666  android.hardware.contexthub@1.0-service
  667  android.hardware.drm@1.0-service
  668  android.hardware.drm@1.1-service.widevine
  669  android.hardware.drm@1.2-service.clearkey
  670  android.hardware.gatekeeper@1.0-service
  671  android.hardware.gnss@1.0-service
  506  android.hardware.graphics.allocator@2.0-service
  504  android.hardware.graphics.composer@2.1-service
  672  android.hardware.health@2.0-service.marlin
  500  android.hardware.keymaster@3.0-service
  673  android.hardware.light@2.0-service
  674  android.hardware.memtrack@1.0-service
  675  android.hardware.nfc@1.1-service
  676  android.hardware.power@1.1-service.marlin
```

Figure 6-16. *A screenshot of my terminal after Frida has been installed*

Next steps will usually involve copying the Frida server onto the Android device, starting the server, and then communicating to the server from the client. This will be covered in a later chapter.

JEB – Android Decompiler

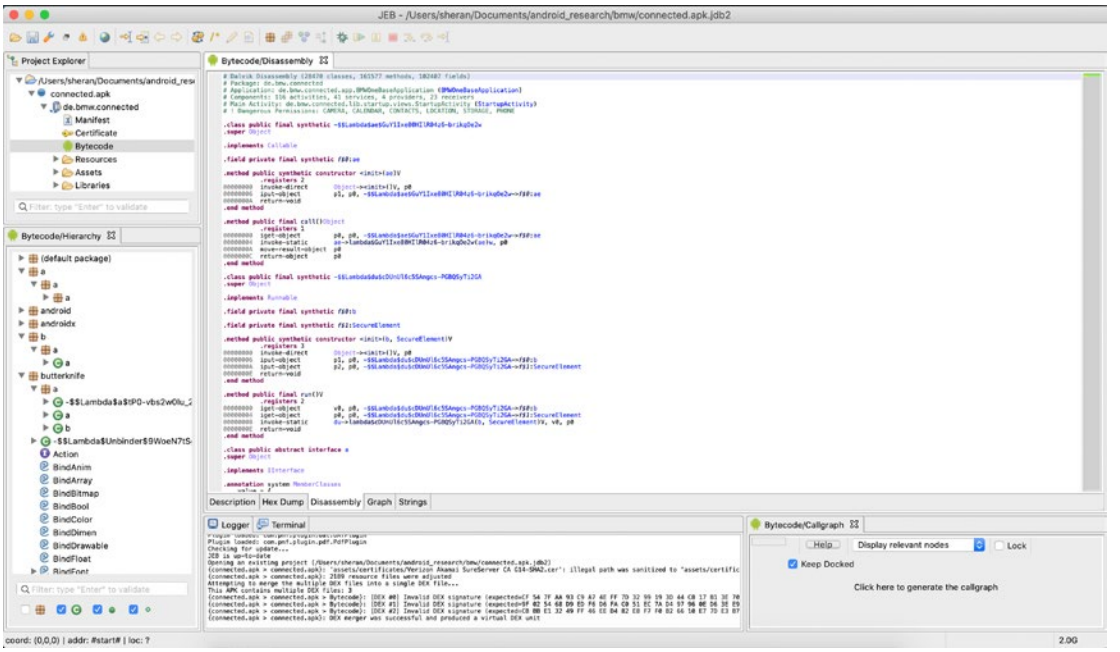


Figure 6-17. *A screenshot from my laptop showing the JEB main screen after loading an APK*

I stumbled onto JEB (www.pnfsoftware.com/) by complete accident when I was researching how I could reverse engineer Android DEX files. In the early days, JEB was usable even if there were some features to be desired. You could use the external scripting module and write your own scripts to improve your workflow or automate tasks. Modern-day JEB is vastly improved with support for disassembling, debugging, and graphing that can show how an app's execution flows.

Now JEB isn't what you would call affordable for the masses. The JEB Pro version that I use will cost you about US $1800 for a 12-month license for one user. That is quite an expensive piece of kit. There is also a JEB Android license that you can get for $1080 per year, or if you prefer, you could get the monthly pay option for $120. While there are some limitations on this edition, it gives you all you need to work on Android decompiling. Now if you are only keen on breaking SSL Pinning or altering the code flow of a specific app, then I think you can manage with Burp, Frida, and the Android emulator or a rooted Android phone. I do, however, think that JEB completes our Tool Bag very nicely, closing off the final gaps in visibility we have of an Android app that we

have not written and have no clue of. JEB can give you direct insight into how the code is written and can even go as far as deobfuscate some obfuscated Android apps. We will touch on this in our other chapters, but first let's take an example where JEB can prove useful.

We saw that with Frida, we have the power of hooking into a specific function; then whenever the Android OS calls that function, we have the ability to replace the existing code in it with our own code and have that run instead. Let's say there was a function in an app that did a comparison. The app execution flow would proceed to the next Android Activity if a specific serial number was provided. Now the comparison function always checks the serial number that was input against a complex internal calculation to verify whether this was a valid serial number. The minute that it detected the serial number that was input was from a valid batch, it would permit access. To bypass this check, we can use Frida to hook into this comparison function. Ultimately, we know that a comparison would yield one of two outcomes: either the numbers are equal or they are unequal. If written in pseudocode, we can envision a comparison function like this:

1. Get input serial number S.
2. Run complex calculation and derive result C.
3. If S could be divided by C with no remainder, then grant access.
4. If S could not be divided by C with no remainder, then return an error.

In the majority of cases, it is nearly impossible to guess the serial number such that it grants us access. Instead, what we can do is to modify the code of the app so that it grants us access even if the wrong code is entered. To do this, we can rewrite this pseudocode to look like this:

1. Get input serial number S.
2. Run complex calculation and derive result C.
3. If S could **NOT** be divided by C with no remainder, then grant access.
4. If S could not be divided by C with no remainder, then return an error.

Note Step 3 and how we changed it so that we would get access even if we provided the wrong serial number. By altering the app flow in this way, and entering an incorrect, possibly random serial number, we can gain access to the new Activity. Since the

majority of cases of guessing involved getting the answer wrong, the app execution flow will never even hit line 4. This is because it will receive the wrong serial number and still grant access to the next Activity.

There is one problem, however. We do not know the name of the function that is calling this Step 3 comparison check. How would we get it? Well there are several different ways that can also include a debugger like Frida where you run the app until the prompt comes up and then pauses the execution so that you can literally step through each line of code and see how the variables of the app change and react. Another way would be to fire up JEB and disassemble the app and look for the code where the comparison takes place. When reverse engineered, it should be possible to more easily and closely locate the function that is of interest to us in altering the execution flow. After this is found, we can directly reference it in Frida and alter the program flow to always grant access even if an incorrect serial number were used.

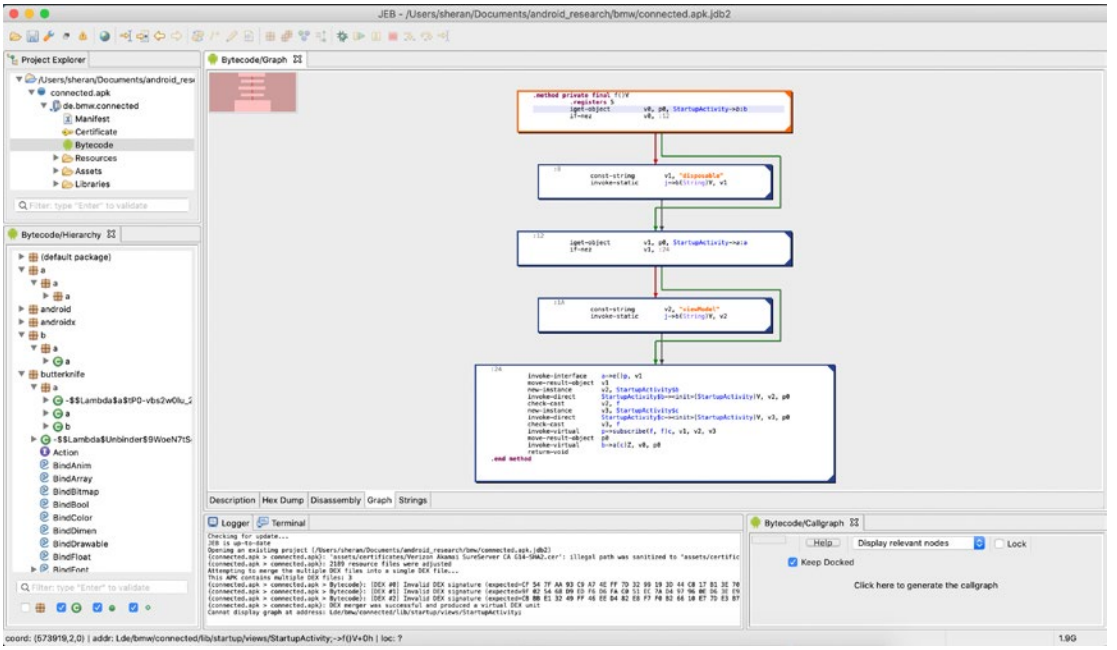


Figure 6-18. *A screenshot from my laptop showing JEB with its graph view of the code*

Some Thoughts on Environment Setup

I think as important as the tools are to our specific use case or requirements, the manner in which you set up your research environment or lab is equally important. In my case, I use the following setup:

- Apple Mac Mini with 32 GB RAM and 1 TB storage
- Two 24” monitors set up side by side in landscape mode
- Mouse
- Keyboard
- Apple MacBook Pro 13” with 16 GB RAM and 1 TB storage
- OnePlus 5T Android phone – rooted
- Google Pixel 3 Android phone – rooted

I use the Mac Mini for my main disassembly and debugging process. I use the MacBook Pro for research or searching for information online or figuring out how something just works.

If I were to call out one of the most helpful features of my setup, I would say it is the two monitors set up as one big canvas. Sometimes when disassembling and debugging in tandem, the extra screen real estate becomes a lifesaver. I can leave my disassembly window as is and shift focus to another debugging process without the pain of switching windows all the time.

CHAPTER 7

Hacking Your App #2

We're going to take apart some apps in this chapter. This time, we will examine what an app looks like while it is running. Stepping through machine code is not my intention in this chapter. Instead, we are going to look at how we can reveal the inner workings of an app by tracing back from its functionality and user interface. A key skill that will unlock this all for us is called dynamic analysis.

Dynamic Analysis

One can argue that dynamic analysis is the same as debugging. It isn't. Debugging usually takes place in a friendly environment. By this, I mean the app is debugged by one or more developers, and there is a lot more information that is available to the person debugging. First is the source code. If you use Android Studio to debug, you have such magical things available to you such as pausing or stepping through each line of code, examining variables, and looking at the debug logs. This is fantastic when you want to trace why displaying that pesky custom-written dialog keeps causing the app to crash. You can set a breakpoint at the line of code just before you show that dialog and then execute the source code one line at a time while studying the variables and debug log file to see what causes the crash.

In dynamic analysis, you're essentially flying blind as far as source code is concerned. Instead, you have access to the bytecode or assembly code. There's nothing essentially wrong with that, but it does take some doing to learn about what your source code looks like in machine code. One thing that I like to do, obviously when I have time on my hands, is to write and build extremely simple Android apps. Something as simple as printing out "Hello World" and then studying what that looks like after it has been disassembled. I find that it gives me a sense of almost unconscious pattern recognition that makes it quick for me to pick out what an app is trying to do in machine code. To further differentiate the debugging element, you are only able to debug your own

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_7

source code. Essentially, you will need to have a debug build that supports things like setting breakpoints and tracing. Generally, the Android apps that you find on the Google Play Store are all release builds which means they have all debugging abilities switched off. If you really wanted to, though, you could enable debugging. Let's see how you can get that done quickly.

Let's fire up the Google Play Store and look for a likely candidate, shall we? Oh, here's one: the NYTimes – Crossword app. I love crosswords. Let's go ahead and download the APK like I've described in Chapter 5. Here's a one-liner that you can use in OS X's terminal:

```
adb pull `adb shell pm path com.nytimes.crossword|cut -f2 -d":"`
./crossword.apk
```

Basically, we're running two adb commands in one go. The inner `adb` statement

```
adb shell pm path com.nytimes.crossword|cut -f2 -d":"
```

will list out the path where the package `com.nytimes.crossword` is installed. Then it pipes that data to the `cut` command which delimits whatever output was piped to it with our delimiter ":" and then takes the second column of that output. What this amounts to is basically the path of the `base.apk` file stored on the Android device. Then all of this is enclosed in backticks (the character below the ~ key on my MacBook Pro keyboard), which means that it gets executed and the result is passed to the outer `adb` statement. The second half of the command tells `adb` to rename the pulled file to `crossword.apk`.

Note that the `com.nytimes.crossword` and `crossword.apk` are dynamic variables that you replace given the situation or APK that you're downloading. Meaning that these two can vary.

Now, we're going to make the APK debuggable. This allows us to run it in a debugger, set breakpoints, and look at the program flow. So to do that, we have to unpack, edit, and repack the APK.

Disassembling the APK

Remember `apktool` that we discussed in Chapter 6? We're going to use that to disassemble the APK we just downloaded.

```
➔  apktool d crossword.apk
I: Using Apktool 2.4.0 on crossword.apk
I: Loading resource table...
```

```
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /Users/sheran/Library/apktool/
   framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Baksmaling classes2.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
➔
```

Setting the "android:debuggable" Flag

Then we need to edit the `AndroidManifest.xml` file. In your favorite text editor, edit the file and look at the `application` section. It should look like this:

```
<application android:allowBackup="false" android:appComponentFactory=
"androidx.core.app.CoreComponentFactory" android:fullBackupContent=
"@xml/appsflyer_backup_rules" android:icon="@mipmap/ic_launcher"
android:label="@string/app_name" android:name="com.nytimes.
crosswordlib.CrosswordApplication" android:resizeableActivity="false"
android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="false"
android:targetSandboxVersion="2" android:theme="@style/AppTheme">
```

We have to add the text `android:debuggable="true"` within that section. So it should look like this after we have edited it:

```
<application android:allowBackup="false" android:appComponentFactory=
"androidx.core.app.CoreComponentFactory" android:fullBackupContent=
"@xml/appsflyer_backup_rules" android:icon="@mipmap/ic_launcher"
android:label="@string/app_name" android:name="com.nytimes.
crosswordlib.CrosswordApplication" android:resizeableActivity="false"
android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="false"
android:targetSandboxVersion="2" android:theme="@style/AppTheme"
android:debuggable="true">
```

Reassembling and Signing the APK

After this, we have to repackage the APK. To do this, we have to build and sign it. If you have already set up your Android Studio with your keystore, then skip this section, but if not, follow along. In Android Studio, start a new Empty Activity project. Then on the Build menu, select `Generate Signed Bundle / APK`.... You should see a window asking whether you want to build an Android App Bundle or APK. Select APK and then you should see a dialog similar to Figure 7-1.

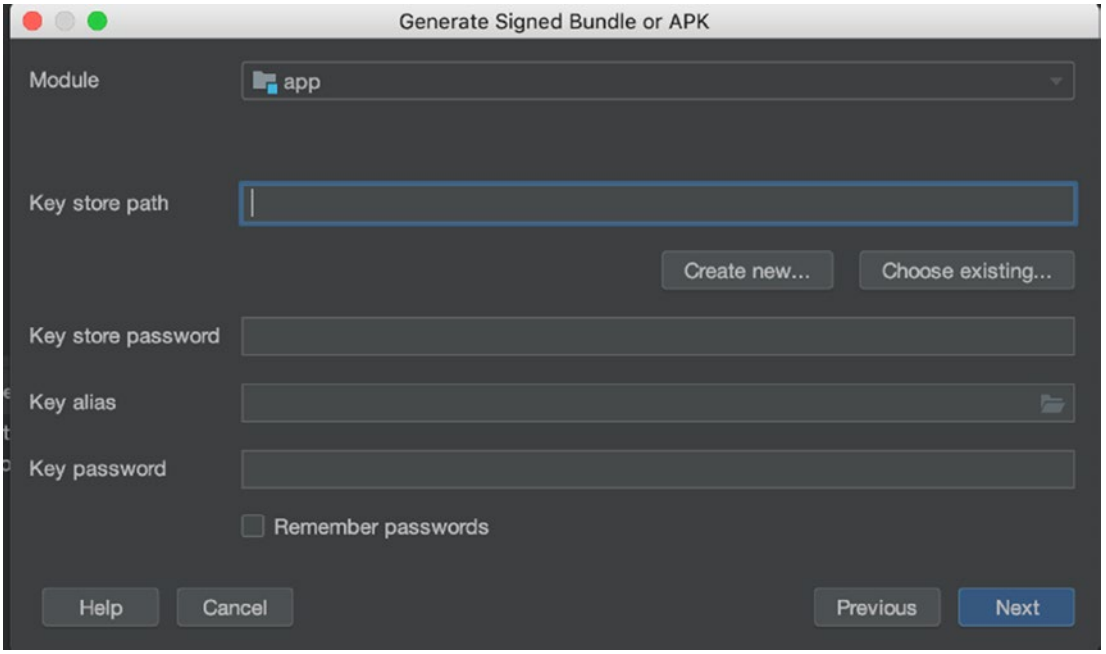


Figure 7-1. *Dialog to create a keystore on Android Studio 3.6*

Next, click the Create new... button and fill in the fields in the subsequent dialog as show in Figure 7-2.

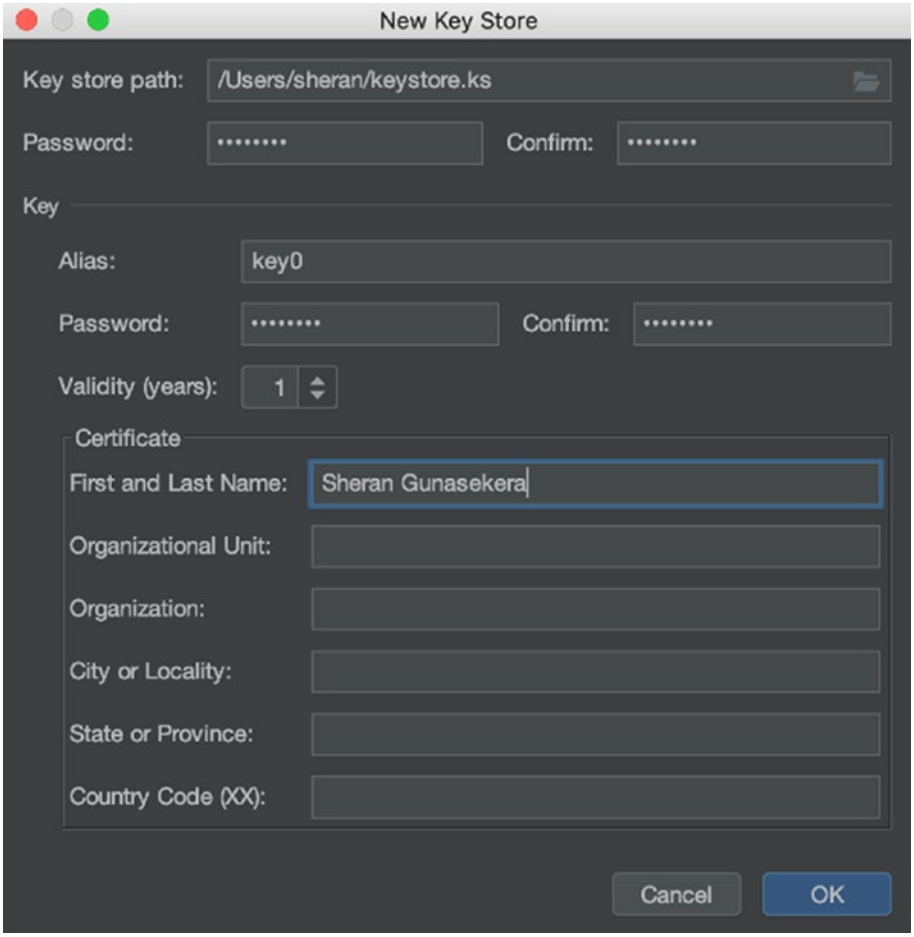


Figure 7-2. *Creating a new keystore*

Make sure to select a path and file name that you remember. Then fill in a password for the keystore and for the private key that we're calling key0. You can rename it if you wish. By default, Android Studio selects a validity period of 25 years; I have brought that down to 1 year as you can see. I have also added my first and last name to the corresponding field. Now click OK. You should be taken back to the prior dialog with the corresponding fields filled in. Click Next to move along. You will be asked to select a Build Variant. Pick debug, then click the V1 (Jar Signature) checkbox, and click Finish.

Android Studio will do its thing and build and sign your APK. It should also build you a nice keystore file in the directory you selected.

Now you may be wondering why the hell I asked you to build an Empty Activity project just so you get the side effect of building a keystore, and you would not be wrong in thinking I'm trying to waste more page space, but let me explain. The alternative is to build the keystore from the command line. Before you do that, however, you will need to have the latest Java Development Kit (JDK) installed.

Install the JDK by visiting this URL: www.oracle.com/java/technologies/javase-jdk14-downloads.html.

You will have to pick the correct installer for your platform. The preceding link contains a link to JDK version 14, which was current as of writing this book.

I'll wait for you to go install it and come back. Done? Awesome. Here's how you generate a keystore file from the command line:

```
➔  keytool -genkey -v -keystore /Users/sheran/keystore2.ks -alias ks2
    -keyalg RSA -keysize 2048 -validity 365
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Sheran Gunasekera
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=Sheran Gunasekera, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown,
C=Unknown correct?
  [no]:  yes
```

```
Generating 2,048 bit RSA key pair and self-signed certificate
(SHA256withRSA) with a validity of 365 days
    for: CN=Sheran Gunasekera, OU=Unknown, O=Unknown, L=Unknown,
    ST=Unknown, C=Unknown
[Storing /Users/sheran/keystore2.ks]
➔
```

Now that we have our keystore, let's rebuild the APK file. We do this by executing the following command on `apktool`:

```
➔  apktool b -o com.nytinmes.crossword crossword
I: Using Apktool 2.4.0
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether sources has changed...
I: Smaling smali_classes2 folder into classes2.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs... (/lib)
I: Copying libs... (/kotlin)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
➔
```

Going back to why I asked you to build the keystore with Android Studio, it's a lot easier – especially if you're still struggling with the JDK. After we have repacked our edited app, we have to sign it. If not, we can't install it on an emulator or device. To sign, we can use either `apksigner` or `jarsigner`. Because you accused me earlier of wasting page space, I'm going to show you both. So there.

Signing with apksigner

apksigner should be installed when you installed the SDK Build Tools from Android Studio. You run it from the command line thus:

```
➔  apksigner sign --ks /Users/sheran/keystore.ks com.nytimes.crossword.apk
```

It asks you for your keystore password and signs the APK.

Signing with jarsigner

`Jarsigner` comes with the Java SDK, and if you set it up correctly, then it will be in your path. To sign with `jarsigner`, you have to type in a bit more:

```
➔  jarsigner -sigalg SHA1withRSA -digestalg SHA1 -keystore /Users/sheran/
    keystore2.ks com.nytimes.crossword.apk alias_name
```

Here, I am using the keys generated from `keytool` which is why you can see me use the alias `alias_name`. Then you can install it on your device or emulator by doing a

```
➔  adb install com.nytimes.crossword.apk
Performing Streamed Install
Success
➔
```

From this point, you can fire up the app in a debugger like gdb, IDA, or JEB. I say gdb, but you can be certain that I haven't used it for at least 10 years now. I stick with JEB. Now admittedly, JEB's debugger isn't the greatest one out there either, but it does exactly what I want it to. JEB Pro and IDA Pro look pretty similar price-wise, but JEB has an Android-only version which is about $1080 per year which is not too bad. Also, if you're idly wondering, I am not getting sponsored by JEB, although it would be kinda cool to get a year of JEB Pro for free. Just saying.

Enough of this nonsense. Let's get back to our app debugging.

Debugging with JEB

Let's continue with the Crossword app that we just changed to enable debugging. Fire up JEB and open the `com.nytimes.crossword.apk` file. JEB works like how you would do a lazy load of images on an ecommerce site. Only load the resources as needed when you get to that point. This is good because when decompiling an APK, it can get pretty big on disk, and it is a lot of stuff to keep in memory. JEB will always tell you if your APK has a lot of resources to process and will ask you whether to load it now. I usually say no to this because it just gets loaded later on anyway when necessary. Figure 7-3 shows this message.

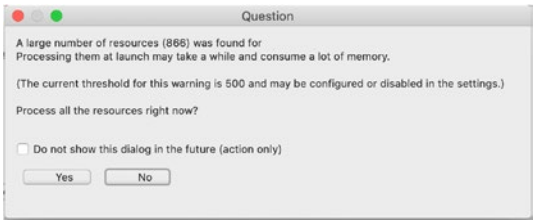


Figure 7-3. *JEB asking whether to load resources*

After JEB processes and loads up the file, you will see the main workspace area. The part that takes the front and center is the bytecode which was just disassembled from the APK. This is Dalvik bytecode which you can find more information about here: `https://developer.android.com/reference/dalvik/bytecode/package-summary`.

On the left panes, you will see the package explorer as well as the hierarchy of how the package is structured within the app. This is handy with which to navigate from. Each left pane also has a "filter" text entry field that you can type in a search string into, and the pane will only display items that have this text. This is very useful for narrowing down classes when moving back and forth within large apps. On the bottom and bottom right, you will find the logging, terminal, and the callgraph generator. I don't use the callgraph generator much, but it allows you to visually look at how specific packages or classes are called and how they fit in with the rest of the code. My JEB layout looks like the one in Figure 7-4.

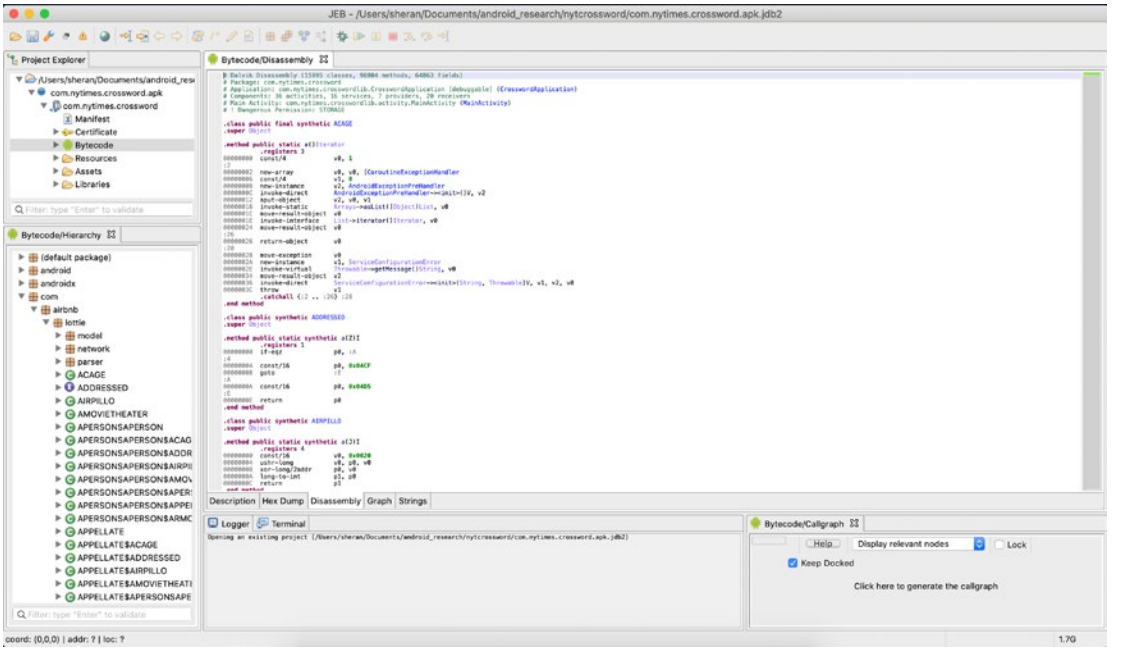


Figure 7-4. *JEB layout with the Crossword app loaded*

The first thing you may notice on the main Bytecode window is the first few lines of comments. I reproduce them here:

```
Dalvik Disassembly (15995 classes, 96904 methods, 64863 fields)
# Package: com.nytimes.crossword
# Application: com.nytimes.crosswordlib.CrosswordApplication [debuggable]
(CrosswordApplication)
# Components: 36 activities, 16 services, 7 providers, 20 receivers
# Main Activity: com.nytimes.crosswordlib.activity.MainActivity (MainActivity)
# ! Dangerous Permission: STORAGE
```

Here, JEB tells you how many classes, methods, and fields there are in total including all third-party libraries. It also tells you that the app is debuggable (yes, we made that happen). Then it tells you the starting activity that loads up when a user opens the app – `MainActivity`. You will notice that the `MainActivity` word is clickable. Most of the classes and methods in JEB are clickable and will take you to the declaration of that specific method or field. This is extremely convenient when navigating static code. I remember in days gone by, your deadlisting of code was just straight up lines of text that you had to scroll through. Generally, we used to print out pages and pages of

disassembled code and make notes on them. While fun, I would never want to go back to those days when disassembling a modern Android app.

When I click the `MainActivity`, it takes me to the bytecode of the start of the `MainActivity`. The first few lines look like this:

```
.class public MainActivity
.super GDPROverlayActivity

.implements EXPERTISE
.implements APERSONSAPERSON
.implements BATTERS$ACAGE
.implements AIRPILLO

.field private D:Z

.field private E:Z
...
...
...
.method public constructor <init>()V
          .registers 3
00000000  invoke-direct       GDPROverlayActivity-><init>()V, p0
00000006  const/4             v0, 0
00000008  iput-boolean        v0, p0, MainActivity->D:Z
0000000C  iput-boolean        v0, p0, MainActivity->E:Z
00000010  iput-boolean        v0, p0, MainActivity->G:Z
00000014  new-instance        v1, Handler
00000018  invoke-direct       Handler-><init>()V, v1
0000001E  iput-object         v1, p0, MainActivity->J:Handler
00000022  new-instance        v1, ACAGE
00000026  invoke-direct       ACAGE-><init>()V, v1
0000002C  iput-object         v1, p0, MainActivity->K:ACAGE
00000030  iput-boolean        v0, p0, MainActivity->L:Z
00000034  return-void
.end method
...
...
...
```

The ellipses are there to spare you of the field declarations and other code. I included some code as well so you can see what the bytecode looks like. I made one line of code bold to show you the format of Dalvik opcode. The first column is the address of the instruction; the second column is the opcode – in this case, `iput-boolean`. The third column contains the operands. So how the `iput-boolean` opcode works is by taking the value in `v0` and putting it into an instance field. In this case, the instance field is `MainActivity->E:Z`. The actual instance that contains this instance field is referenced by `p0`. Now we can take a closer look at what that means by setting a breakpoint on it and then running the program in the debugger. I'll show you that in more detail soon, but for now, let's look at the operands in detail. Figure 7-5 shows the variables being accessed at the time that we stopped on our breakpoint. So, in this case, you can see that `p0` is referencing the `MainActivity`. Essentially, we're setting a `0` into `v0` which resides in the `MainActivity`.

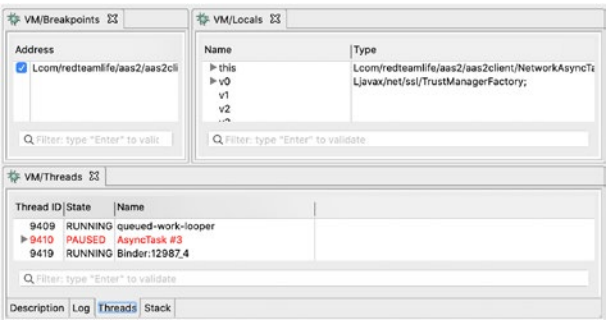


Figure 7-5. *Local variables at the moment our breakpoint was triggered*

With JEB, you can pick a line of bytecode while you're examining it and press the Tab key. This will then invoke the decompiler, and you get back some more readable Java code. Let's look at our routine:

```
public MainActivity() {
        this.D = false;
        this.E = false;
        this.G = false;
        this.J = new Handler();
        this.K = new io.reactivex.disposables.ACAGE();
        this.L = false;
    }
```

Would you look at that? We are setting a `0` (false) into `MainActivity->E` – just as described through the Dalvik opcode.

Now obviously, you may want to browse your code only through the decompiled Java code which is perfectly fine. You can do the same clicking and following around classes, methods, and fields to your heart's content. JEB is seriously useful for Android reverse engineering.

The next set of steps is quite unique to the researcher or hacker. Often it involves trying to solve a puzzle and hit the correct cluster of code that is triggered at a specific action in the app. What does this mean? Well, think about the app that fetches some profile data from a server and presents it to you in a dialog window. If you want to know exactly when the network call was fired, then one way to find that part of the code, especially if the source is obfuscated, is to find the bit of code that pops open the dialog window and then work backward from there. Usually, that involves looking for Android Activity classes. One thing to note about Activity classes is that if mentioned in the AndroidManifest.xml file, the names are left intact by obfuscators like ProGuard. If, in the manifest file, the additional `android:exported="true"` is present, then again the obfuscators will not touch them as they are considered public APIs that must remain callable from outside the app. There are commercial and more complex obfuscators like DexGuard and Arxan that we will cover separately and we can take a look at. These obfuscators handle the obfuscation in a different manner that sometimes even includes manifest files.

Needless to say, this part of the analysis can take a while. Let's turn our attention back to the Crossword app. You will notice that within the app, there is a mechanism to check each square or word that you have entered in the crossword to determine if it is correct. Looking at this, there obviously has to be a way that the answers to the crossword are also delivered or at least fetched. Let's try to make that a goal for this app. Let's try to find out what the answers are to a puzzle. If you look further at the app, you will see that there's a leaderboard and several factors that help you score. One factor is whether you have checked any squares or words. Another is the time. Technically, you could move yourself to the top of the leaderboard by getting all the answers beforehand and then filling them in very fast. So, the goal of this part is to find the routines or the code within the app that gives you all the answers to the crossword. Awesome. Let's do this!

The first thing I want to do is to look through the classes and methods belonging to the app. The first part is to figure out what the app-specific code is. This is among the myriad of third-party libraries that are usually the norm for Android apps these days.

Scrolling through the class hierarchy, I notice two packages: `com.nytimes.crossword` and `com.nytimes.crosswordlib`. I usually browse through each of the packages to see if there are group names that reveal the purpose of that package. Figure 7-6 has a picture of our app package hierarchy. Some interesting packages to examine are the `activity`, `game`, `gamestate`, and `models` ones.

The `activity` package as expected contains all the activities that get launched when you interact with the app. For example, when you click the settings hamburger menu, the `SettingsActivity` is launched, and the setting screen is displayed. Similarly, when you click About, the `AboutActivity` is displayed. Let's make a note to come back to this later and keep looking for more interesting things. The approach that I take is to look through every package first so I can draw up a plan of areas I want to revisit and then prioritize them. So, let's move on.

Figure 7-6. *The app package hierarchy*

The game package has just a few class files, but one looks interesting – `GameChronometer`. Taking a look inside, you see these two methods:

```
@Override  // android.widget.Chronometer
    public void start() {
        super.start();
        this.g = true;
    }
```

```
@Override  // android.widget.Chronometer
    public void stop() {
        super.stop();
        this.g = false;
    }
```

This is most likely where the game timer is started and stopped. That's extremely interesting. We should verify that later. Let's move on for now.

The `gamestate` package looks similar with only a few classes. The most notable one looks like the `ArchiveDataSource` class. It contains some string references to DAOs or Data Access Objects. That may be something we should follow at some point.

That leaves us with the `models` package which is usually named so because they contain data objects or models. Think the model from MVC (model-view-controller). In general, if an app has something related to models, you can hope to find some useful nuggets of information within. Our package is no exception. I see the class named `ClueList` which is, I think, a great place to start our hunt. Wow, didn't even have to get too far to see the reference to a method called `getPuzzleData()`! It returns a `PuzzleData` type object.

```
ClueList(Game arg9, String arg10) {
        List v9_3;
        this.related = ImmutableList.g();
        this.thisListsDirection = arg10;
        int v0 = 0;
        GameResults v9 = (GameResults)arg9.getResults().get(0);
        int v1 = v9.getPuzzleMeta().getWidth();
        PuzzleData v9_1 = v9.getPuzzleData();
        if(v9_1 != null) {
            List v4 = Arrays.asList(new String[]{"Across", "Down"});
            if(v9_1.getClueListOrder() != null) {
                v4 = v9_1.getClueListOrder();
            }
```

Let's trace that. Well, well, well. It seems like `PuzzleData` belongs to the package `com.nytimes.crossword.rest.models`. It's interesting because when you look in there, there's a lot of other model data types that aren't even obfuscated. But get this, look at this snippet of code in the very beginning of `PuzzleData`!

```
package com.nytimes.crossword.rest.models;

import androidx.annotation.Keep;
import com.google.common.collect.ArrayListMultimap;
import com.google.gson.annotations.SerializedName;
import java.util.List;
import java.util.Map;

@Keep
public class PuzzleData {
    private List answers;
    private Map cellOverrides;
    @SerializedName(alternate = {"v6ClueListOrder"}, value =
    "clueListOrder")
    private List clueListOrder;
    @SerializedName("clues")
    private Map cluesMap;
    private Clues cluesWrapper;
    private List layout;
    private MergeAcrossDown mergeAcrossDown;
    private Overlay overlayImage;

    public List getAnswers() {
        return this.answers;
    }

    public Map getCellOverrides() {
        return this.cellOverrides;
```

This is getting more interesting by the minute. You know what we should do? We should set a breakpoint on `getAnswers()` and see what we get! So let's set a breakpoint at the `return this.answers;` line and see what we get. In JEB, you set the breakpoints on the bytecode component. On the decompiled Java code, go to the line where you want to set the breakpoint, hit Tab, and JEB takes you back to the bytecode equivalent of that line. Then you can set the breakpoint. After setting the breakpoint, start the debugger. You will see a dialog that looks like Figure 7-7 which shows you the available devices (in my case, I am running it on an emulator) and then the process id for your debuggable APK.

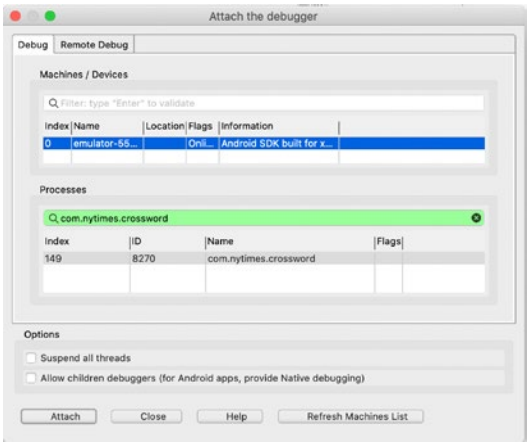


Figure 7-7. *The Attach the debugger dialog window showing which process id to attach to*

One thing to note with the JEB debugger is that you have to have the app that you're debugging launched. If not, JEB won't be able to find it in the running processes. Now that the breakpoint is set, go ahead and click a puzzle to see if our breakpoint gets hit.

Oh my. Would you look at that in Figure 7-8? This is what my JEB screen showed me. On the left, you will see the highlighted line of text next to the circle (which is our breakpoint). The highlighted line is the line of code where the app execution stopped. On the right is the pane that allows you to inspect your breakpoints, local variables, and the threads running. Generally, we're almost always interested in the values section and with good reason. I've gone ahead and expanded the values for you to see. `PuzzleData` contains the List answers. Expanding answers shows us that each item in this list is a `PotentialRebusType` class. Expanding each one, you will see its structure. But more importantly, you will see a list of all the letters in each of the answers. I think there's absolutely nothing wrong for you to take some time and say out loud: Jackpot!

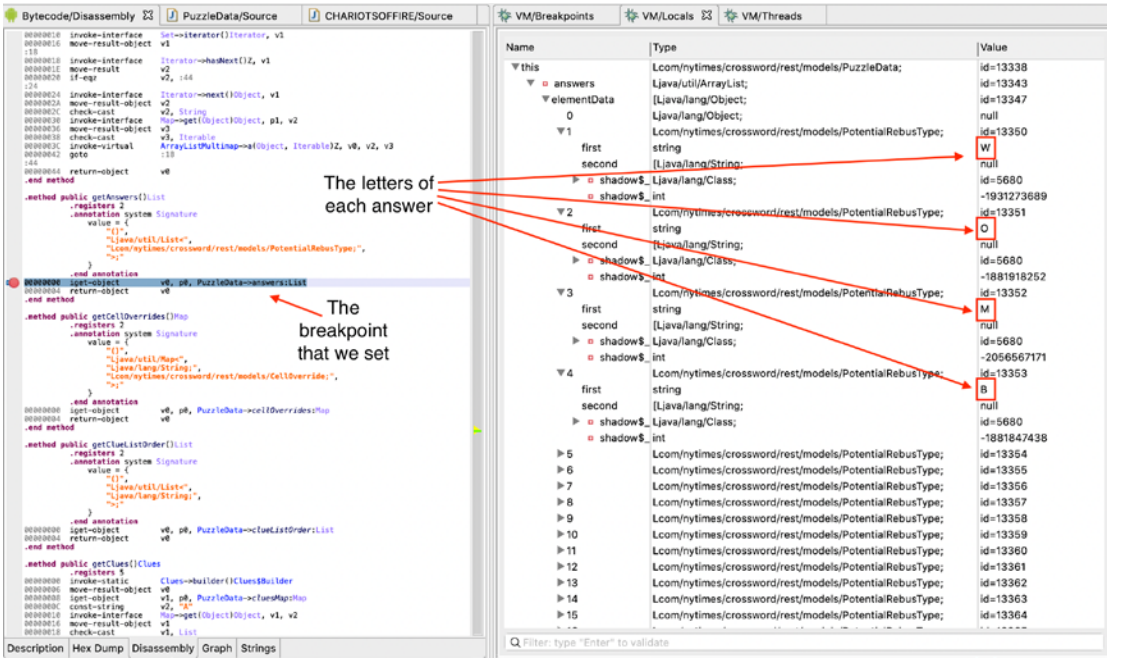


Figure 7-8. *JEB user interface showing local variables and answers from our puzzle*

I try out some of the answers, and sure enough, they work a treat. So, I guess we accomplished our mission of finding out the answers to the puzzle before we attempt it.

Debugging for Free

I know by now you have two questions in your head. Number 1, "Sheran, you're an ass. Why do you have to show us this on a tool that costs a lot of money?" and number 2, "Your solution seems very clunky especially to be able to read the answers off the debugger like that. Can't you pretty print the answers?"

Allow me to answer those questions. For number 1, I anticipated you would ask me this which is why I'm writing this part of the chapter. For number 2, all I can say is: Stop whining! Nothing about app reverse engineering and debugging is pretty. But let's see what we can do all the same.

If you turn your attention back to Chapter 6, you will recall I mentioned a tool called Frida. Frida isn't a debugger per se; it dubs itself as a dynamic instrumentation toolkit. Basically, it can do a lot of cool stuff, and if you ever meet Ole André V. Ravnås, the author of Frida, you should buy him a beer. Hell, buy him a whole bottle of Scotch.

Again, in Chapter 6, I showed you how to install the Frida tools onto your workstation. The next step that we have to get done is to install the Frida server on the mobile device. Let's use our emulator for this. If you recall, I showed you how to set up Android Studio and create a virtual device using the AVD manager. We're going to need a virtual device for this exercise. Go and create a virtual device that can run root commands. Any image that doesn't have the Google Play Store enabled will do. I created a Pixel 2 XL device running API version 27. Once you get that done, fire it up and install the Frida server. I download the prebuilt version of the Frida server, and you can find it here: https://github.com/frida/frida/releases. You will have to get the version that matches your hardware. In this case, since we're running our emulator on our laptop, you will need to get the x86 version. The one we want is `frida-server-12.8.20-android-x86.xz`. I use a plain old `wget` to download the file thusly:

```
wget https://github.com/frida/frida/releases/download/12.8.20/frida-server-
12.8.20-android-x86.xz
```

You can download directly from the browser if you wish. Next, to decompress the file, use

```
gzip -d frida-server-12.8.20-android-x86.xz
```

This should leave you with the decompressed file in your current directory like this:

```
➔  ls -al
total 51008
drwxr-xr-x   3 sheran  wheel        96 Apr 28 21:07 .
drwxrwxrwt  38 root    wheel      1216 Apr 28 21:05 ..
-rw-r--r--   1 sheran  wheel  26114852 Mar 31 01:41 frida-server-12.8.20-
android-x86
➔
```

To make things easier, I rename my file to `frida-x86` and then copy it over to the emulator:

```
➔  adb push frida-x86 /data/local/tmp
frida-server-12.8.20-android-x86: 1 file pushed, 0 skipped. 145.3 MB/s
(26114852 bytes in 0.171s)
➔
```

And then, connect to the emulator to run the Frida server:

```
➔  adb shell
generic_x86:/ $ su
generic_x86:/ # cd /data/local/tmp
generic_x86:/data/local/tmp # chown root:root frida
generic_x86:/data/local/tmp # chown root:root frida-x86
generic_x86:/data/local/tmp # chmod u+x frida-x86
generic_x86:/data/local/tmp # ./frida-x86 &
[1] 9434
generic_x86:/data/local/tmp #
```

What we're doing line by line is connecting to the emulator, then switching over to the root shell.

If your `su` command fails, then you may have to restart your `adb` server to support switching to the root user. Execute `adb root` first before you try to use `adb shell`.

Then we change the owner for the `frida-x86` file and make sure we change the mode of the file to be able to execute. Lastly, we execute it and send it to the background with the ampersand. You can verify that the Frida server is running by doing a `ps` command:

```
generic_x86:/data/local/tmp # ps |grep frida
root          9434  4404   86036  48068 poll_schedule_timeout e9a01af0 S
frida-x86
generic_x86:/data/local/tmp #
```

We can test if Frida server is running fine on the emulator by starting up Frida and attaching to the Crossword app. We can then run some commands to show the process id, for example, or the current platform it is running on. Frida has command completion which is really useful. For the full set of JavaScript API calls that you can execute on the console, see here: https://frida.re/docs/javascript-api/.

As a quick check to see if you have got everything up and running, you can run

```
frida-ps -U
```

And if all is set up well, you should be able to see the process list of the emulator. The output looks similar to the previous chapter's Figure 6-16.

```
(p3) ➔  frida -U -f com.nytimes.crossword
     ____
    / _  |   Frida 12.8.20 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://www.frida.re/docs/home/
Spawned `com.nytimes.crossword`. Use %resume to let the main thread start
executing!
[Android Emulator 5554::com.nytimes.crossword]-> Process.id
613
[Android Emulator 5554::com.nytimes.crossword]-> Process.platform
"linux"
[Android Emulator 5554::com.nytimes.crossword]->
```

Frida's Interesting Tricks

I would have liked to title that Frida's Interesting Trick, because frankly it does one thing really well which is to allow you to hook functions either native or Java. It does a lot more than just that, but I will now show you how we can use exactly this technique to do something fun with this Crossword app.

What we're going to do is to hook the Java `HashMap` class' `put()` method to intercept the data that is being placed into a `HashMap`. Why do this? Let me show you in Figure 7-9. You will notice that we're looking at a class called `CHARIOTSOFFIRE`. This class is initiated with the `Game` object. Within this `Game` object lies not just the clues to the crossword but

also the related answers. You will see the line of code that I have highlighted is the one where the answer is put into a `HashMap`. If we can successfully hook the `HashMap's` put method, then we can see each key and value that is being stored by this line of code.

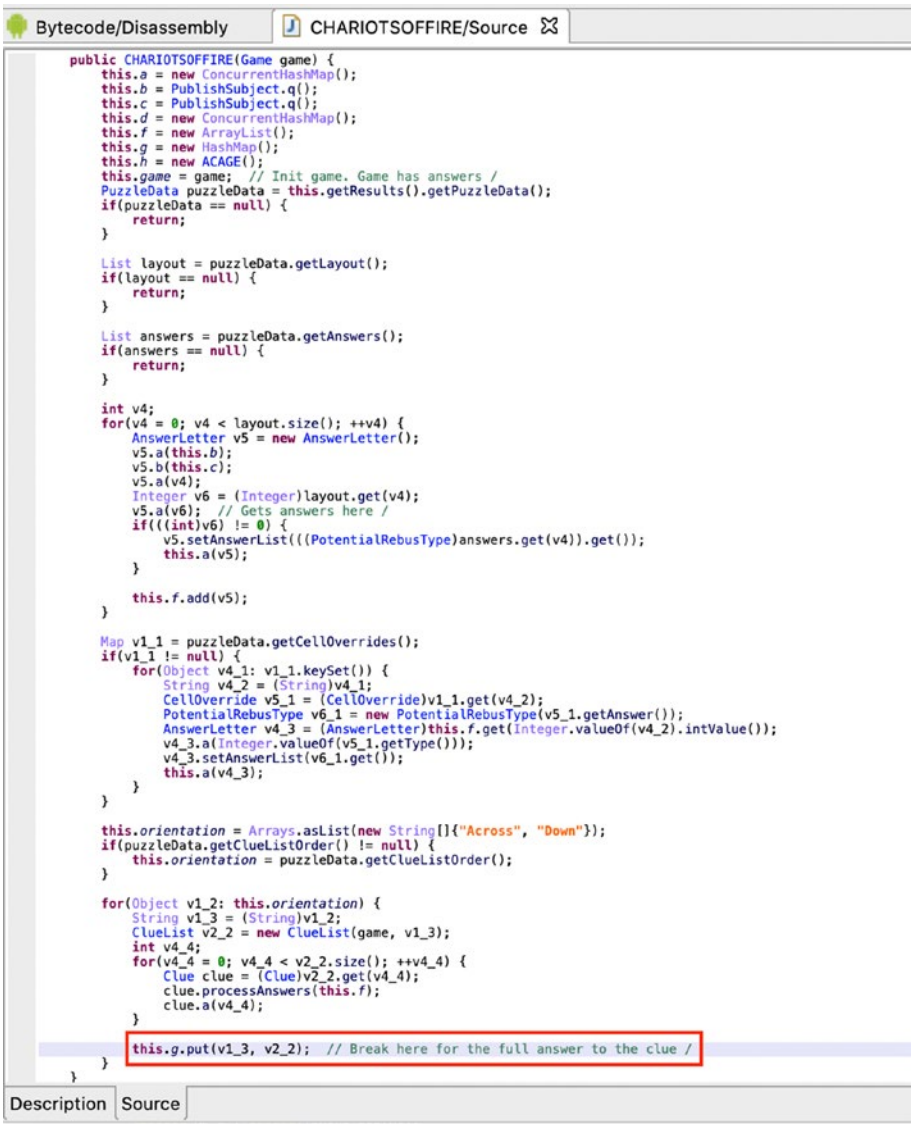


Figure 7-9. *The CHARIOTSOFFIRE class that shows when the answer is written to a HashMap*

We can hook functions using Frida in many ways. I will show you how to do it using Frida and a bit of JavaScript code. Let's go over the code first:

```
Java.perform(function(){

        const hashMap = Java.use('java.util.HashMap');
        const clue = Java.use('com.nytimes.crosswordlib.models.
        BROKENPROMISE');
        const clueList = Java.use('com.nytimes.crosswordlib.models.
        ClueList');
        const arrayList = Java.use('java.util.List');
        const javaStr = Java.use('java.lang.String');
        const answerLetter = Java.use('com.nytimes.crosswordlib.models.
        CONNOTES');
        const clazz = Java.use('java.lang.Class');

        hashMap.put.overload('java.lang.Object','java.lang.Object').
        implementation = function(k,v){
               if ((v != null) && (v.$className == 'com.nytimes.
               crosswordlib.models.ClueList')){
                      var vClueList = Java.cast(v,clueList);
                      var it = vClueList.iterator();
                      while(it.hasNext()){
                             var cl = Java.cast(it.next(),clue);
                             var clueText = cl.c();
                             console.log(clueText);
                             var ans = Java.cast(cl.h(),arrayList);
                             var iit = ans.iterator();
                             var finalAns = [];
                             while(iit.hasNext()){
                                    var ans = Java.cast(iit.
                                    next(),answerLetter);
                                    var ansLetterClass = Java.cast(ans.
                                    getClass(),clazz);
                                    var c_field = ansLetterClass.
                                    getDeclaredField('c');
```

```
26:                 c_field.setAccessible(true);
27:                 var cArrayList = Java.cast(c_field.get(ans),arrayList);
28:                 finalAns.push(Java.cast(cArrayList.get(0),javaStr));
29:                             }
30:                             console.log(finalAns.join(''));
31:
32:                         }
33:                 }
34:                 return this.put(k,v); // Execute the actual HashMap.put()
                    function
35:         }
36:
37: });
```

This script, which we saved as `intercept.js`, hooks the `put()` method of the HashMap class. We first instantiate the `HashMap` class in this script using Frida's `Java.use()` method (line 3) and then overwrite the implementation of the `put()` method in line 11. This means that we now control the `put()` method. The first thing I want to ensure is that we do not interrupt how the actual `HashMap.put()` method works. So we execute the default `put()` method in line 34. OK, now we must find the `HashMap.put()` method that is called with the value type of `com.nytimes.crosswordlib.models.ClueList`. We check that in line 12. Then, we proceed to iterate over the `ClueList` to find `BROKENPROMISE` objects, which I have named as `Clue` objects. Within `Clue` objects are the answers and the text of the clue. This is what we're after. Since the app deals with crosswords, the answers themselves will not be stored as full strings. Instead, they are stored to mimic how letters would look on a crossword and thus are stored a character at a time usually in a Java `ArrayList`. The `h()` method that I call in line 19 is actually equivalent to a `getAnswers()` method. Once again, I have to iterate over the answers because, remember, they are stored as a list of characters; in this case, they are stored as a list of `CONNOTES` objects which I named as an `AnswerLetter` (line 23). This object then contains another `ArrayList` of the letter of the answer. The letter seems to be always stored in the first element of the `ArrayList`. To get this `ArrayList`, we have to access a private field called `c`. I had to do some Java trickery to get the field, because there were two other methods also named `c` which made things ambiguous. This happens after code obfuscation. This happens on lines 25–28.

This is all an elaborate way of saying: Watch the `HashMap.put()` method, and when a list of clues is seen, take note of the clue text and the answers and print them out to the console. Simple enough right? Well, let's find out by running it.

```
(p3) ➔  frida -U -l intercept.js -f com.nytimes.crossword --no-pause
```

This tells Frida to connect to a USB device, in this case, the emulator. Then it says execute the code in the `intercept.js` file as soon as you attach to the process. Frida usually always pauses the process as soon as it starts, so we turn off the pause functionality. We then select a crossword to play, and the script begins to execute. The results are pretty cool and can be seen in Figure 7-10.

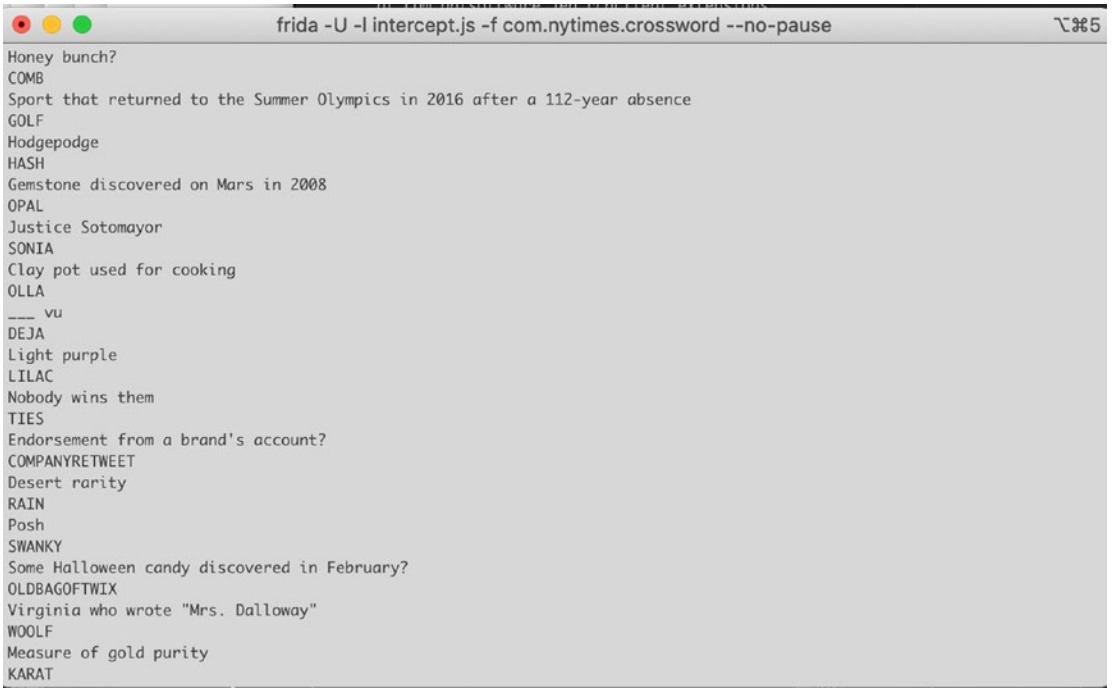


Figure 7-10. *Frida running with our intercept.js script*

What we can see is the clue text followed by an all caps version of the answer to that clue – all printed out even before you attempt the crossword. Fantastic! Now all that's left is to play the crossword with our answers. See Figures 7-11 and 7-12. I averaged almost 4 minutes to play a puzzle with about 60 clues. Slow, I know, but my excuse was that I was playing on an emulator and I had to use the emulator keyboard and not my own hardware keyboard.

Figure 7-11. *Completed 60-clue puzzle in under 4 minutes*

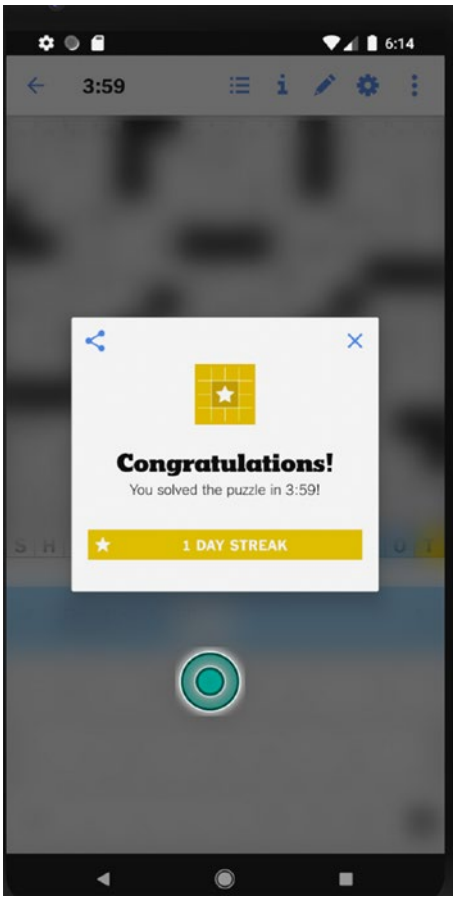


Figure 7-12. *Victory is mine!*

Where do we go from here? Well, the sky is the limit of course. We can play around with the chronometer and basically stop the clock, do the puzzle, then start it up again to post some ridiculously implausible time. I highly recommend that you purchase and subscribe to the NYTimes Crossword puzzle app. It's really great to pass the time with.

We've seen how we can debug (kinda) an app for free using Frida. We also figured out how to print the answers in a marginally prettier manner. As for the deadlisting of code, you can generate that with some of the free decompilers out there like JADX, dex2jar together with JD-GUI, or even Android Studio. But it is essential that you have some form of view of the code as well to make things smoother. You could possibly do it with only Frida, but I think this may take you a little longer to finish up as you may have to trial and error your way through. Some other cool things you can do would be to hook network calls so that you can see traffic going back and forth between the app and the

server. Essentially, if you can think it up, then you can most likely get it done using JEB and Frida. A word to the wise: Don't be mean and don't be destructive. App developers spend considerable time, effort, and even money writing apps. We should be respectful in our use of apps (except for those goddamned spyware-laden ones), and if we like an app and find it useful, then we should always pay for it.

CHAPTER 8

Rooting Your Android Device

You may have heard about "rooting" your device. You've already probably done it and are enjoying its benefits. In this chapter, we will take a look at what it means to root your device, what benefits you can get out of it, and how to actually root a device. We also take a look at some of the things we normally do on an emulator, but this time on the device proper.

The Internet is full of forums where hapless users bemoan their failed efforts at rooting. The perils of a bad rooting session could mean the infamous "bricking" of your device, essentially rendering it completely useless save for securing loose papers on your desk during a windy day where you've left your windows open. I insist that if you wish to follow along with our rooting exercise, you do it on a phone that is not your daily driver where you save all your data. If you have the means and are so inclined, I recommend you pick up a cheap used Android phone and use that instead. I'll talk you through the steps on how I rooted my Google Pixel XL in this chapter. The process should be the same for other devices as well. Samsung's will be trickier, and I do not recommend that you use my technique if you have a Samsung. I will expand on the rooting processes as I gather more data. I will make these available on my companion site to this book which is www.aas2book.com.

What Is Root?

Root refers to the Linux or Unix administrator account. Often called the superuser, the root user is able to modify all parts of a Unix system. It has the highest access rights and allows a user to do things like create and delete other users, read and change any part of the filesystem or configuration, install and remove software on the system, and view all

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_8

network traffic and process-related information on that system. Essentially, if you have root on a Unix system, you can do anything to it as the root user.

As we know, Android is based on Linux and adopts most parts of the Linux kernel. Therefore, it is able to run a host of Linux-based programs and services. The Android shell that you access by typing in adb shell is known as the MirBSD Korn Shell, also called `mksh`. This continues to be the default shell that was shipped from Android 4.0 onward. Through this shell, you can run Linux commands like ls, cd, cat, echo, and so on. In essence, the Android shell looks and feels as if you are in a Linux machine. By default, the root user and access to it has been restricted by the Android system. Thus, whenever you type in adb shell, you get placed into the default user called shell with user id 2000. You can find this information by typing in the Linux id command:

```
1|marlin:/ $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),
1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),
3003(inet),3006(net_bw_stats),3009(readproc),3011(uhid) context=u:r:shell:s0
marlin:/ $
```

On a Linux system, to switch to the root user, you can invoke the su command which is called the substitute user command. If you know the root password, invoking su will first prompt you for the root password and then drop you into a root shell. You will know you are in a root shell because the prompt usually changes to a #.

Why Root?

There are many reasons you may want to root your device. Some that come to mind include the desire to customize your device how you want to. For example, if you're not satisfied with how the system fonts look, or how nav buttons look, then you can easily change that when you have root on your device. Similarly, you may want to remove OEM bundled system applications that take up unnecessary space on your device. You may want to take a look at the data that apps are writing to the file system, or you may want to debug your system. There are also many malicious reasons why someone would want to root an Android device. Some of the most popular reasons are for downloading and running cracked software without paying for it or watching content that has DRM or digital rights management encryption on it – pirating.

We will use root for debugging and security testing. Because Android is based on Linux, it can use many of the Android debugging tools. We will also use root to take a deeper look into the data that each app stores in its own space. Android separates apps by user account. This means that each app will get its own user id and group id and storage directory within the /data partition. As is common with Linux, one user may not see another user's files without explicitly being granted permission to. If you do a long listing in Linux (`ls -al`), you will see the file permissions clearly visible.

```
drwxrwx--x 3 shell shell    4096 2020-05-22 00:46 .
drwxr-x--x 4 root  root     4096 1970-01-01 13:06 ..
drwxrwxrwx 4 shell shell    4096 2020-05-22 00:46 .studio
-rwxrwxrwx 1 shell shell  190464 2017-03-30 22:24 jtrace64
-rwxrwxrwx 1 shell shell   10432 2017-03-30 22:45 plugin.so
-rwxrwxrwx 1 shell shell 5614728 2020-05-16 19:53 strace
-rw-r--r-- 1 root  root  5221990 2020-05-16 23:07 symbols
-rw-r--r-- 1 root  root  1002367 2020-05-16 19:54 xxx
-rw-rw-rw- 1 root  root   506465 2020-05-16 17:45 xxxx
```

The first column in the preceding directory listing shows you the permissions of each file. The format is like this:

```
- r w x r w x r w x
```

The first character indicates if the entry is a directory. If it is, it will have a d in the first character space. The next three characters tell you if the owner and creator (in the third column) of the file can read (r), write (w), or execute (x) the file. The next three tell you the access permissions of the group (in the fourth column). The last three characters tell you what permissions others or "the world" has. If you look at the preceding entry called symbols, you will see that the owner of the file, root, has read and write access to the file. The root group has read, and the world also has read access to the file.

By setting only permissions of the owner of the file to read and write and disabling all other permissions, it is impossible for another user to view or access that file. This is exactly what Android has done to keep app-specific data separate. Each app will have a user and group id and have its data owned only by that user id. Other apps will have their own permissions and will not be able to access each other's data. Here is what it looks like when we view the /data/data partition (where Android stores all its app-related data) for the two apps that we wrote:

```
marlin:/data/data # ls -al com.redteamlife*
com.redteamlife.aas2.aas2client:
total 80
drwx------   8 u0_a178 u0_a178        4096 2020-05-20 18:33 .
drwxrwx--x 229 system  system        20480 2020-05-22 00:46 ..
drwxrws--x   2 u0_a178 u0_a178_cache  4096 2020-05-17 15:36 cache
drwxrws--x   2 u0_a178 u0_a178_cache  4096 2020-05-17 15:36 code_cache
drwxrwx--x   2 u0_a178 u0_a178        4096 2020-05-20 18:33 databases
drwxrwx--x   2 u0_a178 u0_a178        4096 2020-05-20 18:33 files
drwxrwx--x   2 u0_a178 u0_a178        4096 2020-05-20 18:33 no_backup
drwxrwx--x   2 u0_a178 u0_a178        4096 2020-05-20 18:33 shared_prefs

com.redteamlife.aas2.aas2obfuscate:
total 80
drwx------   8 u0_a187 u0_a187        4096 2020-05-25 23:47 .
drwxrwx--x 229 system  system        20480 2020-05-22 00:46 ..
drwxrws--x   2 u0_a187 u0_a187_cache  4096 2020-05-22 00:46 cache
drwxrws--x   2 u0_a187 u0_a187_cache  4096 2020-05-22 00:46 code_cache
drwxrwx--x   2 u0_a187 u0_a187        4096 2020-05-25 23:47 databases
drwxrwx--x   2 u0_a187 u0_a187        4096 2020-05-25 23:47 files
drwxrwx--x   2 u0_a187 u0_a187        4096 2020-05-25 23:47 no_backup
drwxrwx--x   2 u0_a187 u0_a187        4096 2020-05-25 23:47 shared_prefs
marlin:/data/data #
```

The two app bundle names are `com.redteamlife.aas2.aas2client` and `com.redteamlife.aas2.aas2obfuscate`. You will notice that aas2client has user and group id u0_a178, and aas2obfuscate has user and group id u0_a187. Notice that the world or other users can only execute (x) or list the directory of that app. They cannot actually read or write to those directories and files. As root, this restriction is lifted, and you can look in every app's files and directories. We should pause here and understand why it is a bad idea to store any sensitive data like API keys and back-end system passwords on the filesystem of your app. If the owner of the phone roots his device, then he can go into your app data directory and read all the files that you have written.

Sometimes, you may want to run, test, and debug apps that do not run on the emulator. By rooting your device, you can run and debug these apps on your device.

You may also want to do malware analysis, and in cases where malware can detect and refuse to run in an emulator, you can run them on a rooted device. I don't recommend doing this unless you have set the device up properly where the malware cannot escape your device and network and proceed to wreak havoc elsewhere.

Rooting Safely

Unless you are very well aware of Android internals, you will have to rely on a third party to help root your phone. Here, again, it is important to know how your third-party app will root your device. Downloading root apps willy-nilly can get you into hot water, especially if the author has embedded some malicious backdoor code that allows him to return to your device whenever he wants and gain full control of it. The most popular mechanism of rooting that is out there presently is called Magisk. I will take you through the rooting process that Magisk uses. Magisk is an open source project which you can build yourself. You have the freedom to inspect the code to see exactly what it is doing and, therefore, can gain some reasonable confidence that no malicious intent exists. You can find Magisk here: `https://github.com/topjohnwu/Magisk`.

As I stressed earlier, I highly recommend getting a separate device to test your root first. You can of course use an Android emulator, but in some cases, there are apps that will verify you are running on an emulator by doing simple things like fetching the default Bluetooth adapter which will always be undefined or null if run on an emulator. If you absolutely have to root on your own device, then it goes without saying that you need to take a backup of all your data. The rooting method I outline will wipe your data partition anyway, so a backup is essential.

The Rooting Process

The rooting process with Magisk is quite straightforward. Magisk was developed by John Wu who ironically now works for Apple. He still actively maintains Magisk, though, and it is quite an elegant set of tools he has put into place. Here is an overview of what we will do to root our device:

1. Download the matching AOSP (Android Open Source Project) images that match our device
2. Install Magisk Manager on the device via adb

3. Extract the boot image from the AOSP image
4. Patch the boot image through Magisk on the device
5. Unlock the bootloader
6. Flash the boot image that we patched onto the device

For the purposes of this chapter, the device I will use is the Google Pixel XL. Its codename is marlin, and it is running the latest OS version 10.0.0 (QP1A.191005.007.A3, Dec 2019). I will not be able to cover other devices' rooting in this chapter. I do intend to test other devices with root and upload the write-ups to a companion website for this book [https://aas2book.com].

Getting the Factory Image

Android factory images can be found here: https://developers.google.com/android/images. The images range from the Nexus S 4G all the way up to the most recent Pixel 4 and Pixel 4 XL. It is important that we match the factory image to the device. You can always check the phone to see what factory image you would need to download. Go into Settings ➤ About Phone. Then look for both the Model and the Build Number (Figure 8-1). If your phone is another make or model, you will have to check with that phone manufacturer's website for factory images to download. Additionally, there are different mechanisms when it comes to rooting Samsung and Huawei phones. Both are covered on Magisk's website [https://topjohnwu.github.io/Magisk/install.html].

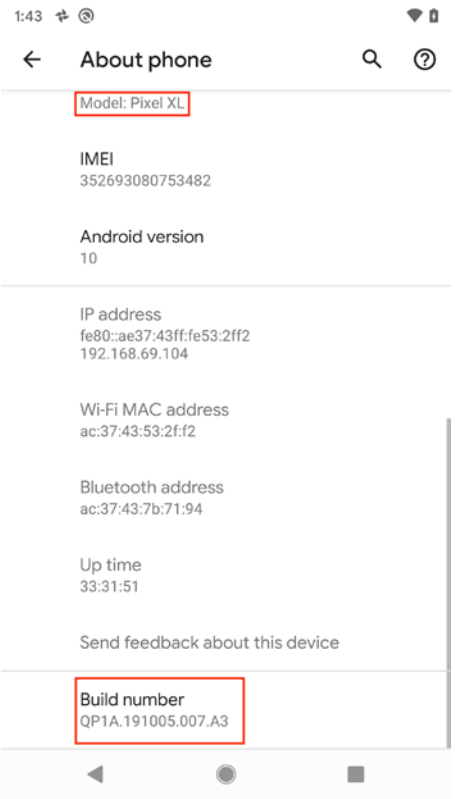


Figure 8-1. *Getting the build number and the device model*

It is definitely worth noting that the factory images for the device are over 1 GB in size so you will have to spend some time downloading them and also need the space to store them. Hopefully, this isn't an issue in the present day, but you have been warned – especially when you realize that you only need one small file from the image. For my device, I downloaded the appropriate version of my image which amounted to 1.4 GB. It came as a ZIP file which I decompressed as shown in Figure 8-2.

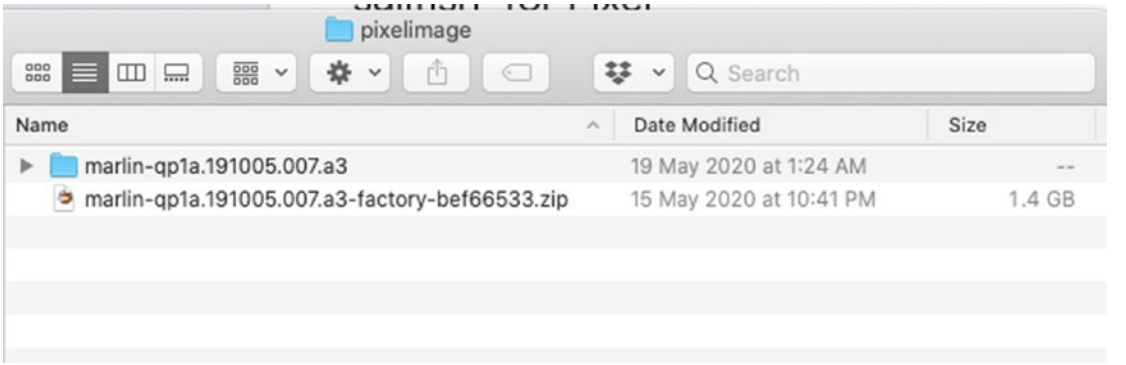


Figure 8-2. *The downloaded image and location of the files when decompressed*

Inside the folder, you will see one more ZIP file. Using my specific case, I see the file named image-marlin-qp1a.191005.007.a3.zip. Decompress this file as well, and you should then see another folder that contains the file called `boot.img`. This is the file that we're interested in. The structure of the folders is shown in Figure 8-3.

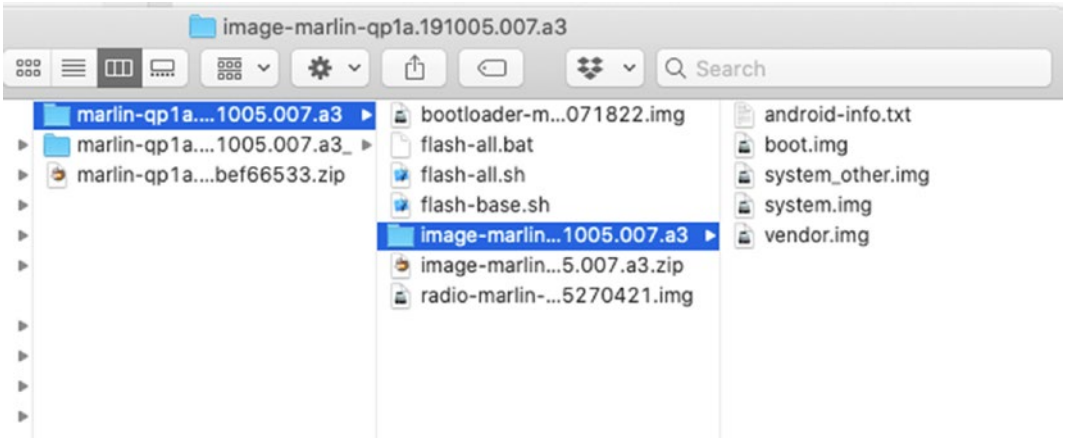


Figure 8-3. *The decompressed structure of the factory image*

Installing Magisk Manager

Magisk Manager is no longer available on the Google Play Store. Instead, you have to get it from the GitHub site [`https://github.com/topjohnwu/Magisk/releases`]. Look for the release named Magisk Manager and download that. At the time of writing this chapter, version 7.51 was released. I will download the APK file now in Figure 8-4.

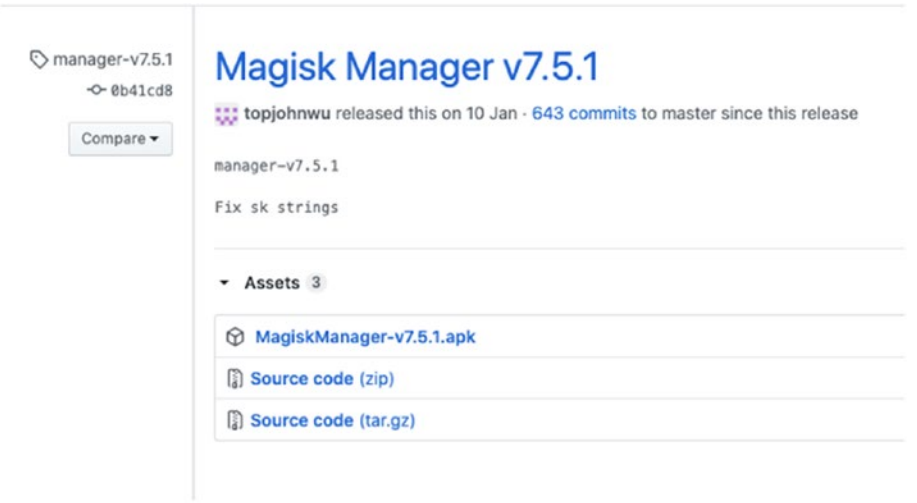


Figure 8-4. *Downloading Magisk Manager v 7.51*

Once downloaded, you can go ahead and install it by running adb install:

```
➜  adb install MagiskManager-v7.5.1.apk
Performing Streamed Install
Success
➜
```

Patching the boot.img File

When installed, you should see the icon in your app screen. Before running it, let's copy over the boot.img to the device so that we can patch it. Go to the directory where you decompressed your boot.img file and then copy it across to the device. We will copy it straight into the Download directory of the device:

```
➜  adb push boot.img /storage/emulated/0/Download
boot.img: 1 file pushed, 0 skipped. 29.8 MB/s (31712486 bytes in 1.013s)
```

Verify that the `boot.img` is now visible on your device (Figure 8-5).

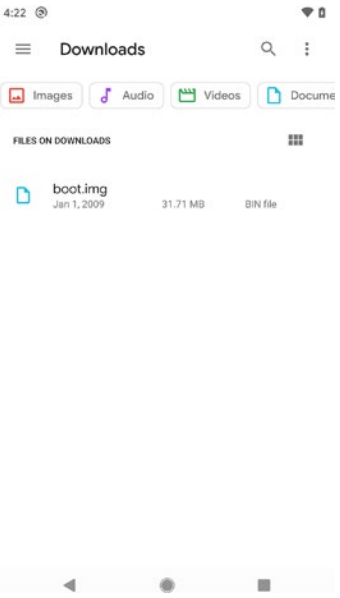


Figure 8-5. *The uploaded boot.img file on the device*

Let's now patch the boot.img file using Magisk Manager. Fire it up and you should see something like Figure 8-6 greet you. Under the Magisk entry, click Install, then click Install again when the dialog window opens, and lastly choose Select and Patch a File (Figure 8-7).

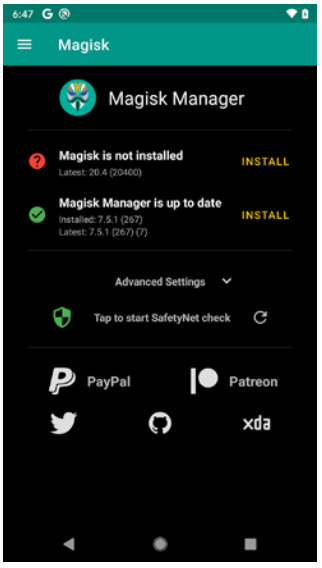


Figure 8-6. *The Magisk Manager main screen when Magisk isn't installed*

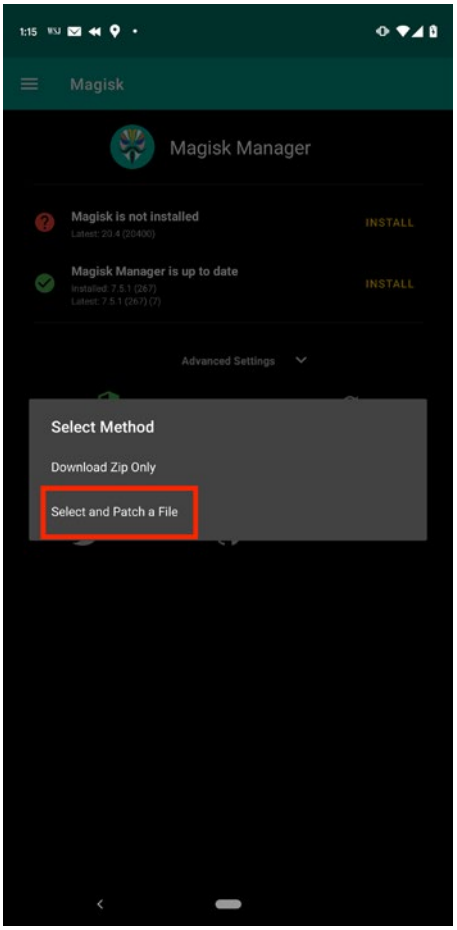


Figure 8-7. *Selecting which method to use when installing Magisk*

The standard file browser should pop up allowing you to select the `boot.img` file to patch. If you get prompted for permissions to access the storage, click Allow. Find the `boot.img` file that you copied over previously and select it. Magisk should then patch the `boot.img`, and when it is done, you should see something like Figure 8-8.

Figure 8-8. *boot.img has been successfully patched*

Now, if you navigate back to your Download folder on your device, you should see a new file called `magisk_patched.img`. This is the patched image file that we will flash onto the device in order to get root. First copy it over from the device back to your workstation using adb:

```
➜  adb pull /storage/emulated/0/Download/magisk_patched.img
/storage/emulated/0/Download/magisk_patched.img: 1 file pulled, 0 skipped.
33.8 MB/s (31937832 bytes in 0.902s)
➜
```

Unlock the Device Bootloader

Android devices have several partitions on their filesystems. Each partition will perform a specific function. The Android bootloader contains a set of instructions that help the device find and boot up the Android kernel. It contains a very minimal user interface and USB interface that allows you to interact with it (as you will soon see). It also permits

users to flash further or additional partitions onto the device. Generally, the bootloader is always locked by the vendor of a device. This is to prevent users from arbitrarily flashing additional or different partitions onto the device to alter their behavior (typically rooting). The bootloader effectively offers up some level of protection (almost like write protection) where you can't either accidentally or purposely flash images onto the device. Some vendors allow you to unlock the bootloader (which you need to flash our modified `boot.img` file); some do not. You may thus read about specific brands of phones being ideal candidates for rooting. These ideal candidates typically allow easy unlocking of their bootloader. In the absence of bootloader unlocking, the next available mechanism of gaining root access is to rely on a known vulnerability that gets you root by exploiting a weakness in the code to elevate privileges. These do not stay active for very long, however, as vendors typically find and patch these flaws in subsequent versions of their Android updates. On the Google Pixel range of phones, the bootloader is very easy to unlock as you will see. First, we have to reboot our device into the bootloader mode. Then we issue a command to unlock the bootloader itself.

As a matter of security, since an unlocked bootloader effectively gives an attacker much easier access to compromise a device by bypassing a security pin, whenever a bootloader is locked or unlocked, it wipes the entire data partition to safeguard the user's private data.

PROCEEDING WITH THE BOOTLOADER UNLOCK IN THIS NEXT SECTION WILL WIPE ALL YOUR DATA!!! MAKE BACKUPS!

With your device still connected to your workstation via USB, issue this `adb` command:

```
➜  adb reboot bootloader
```

Your device should now restart, and you should see a screen similar to Figure 8-9.

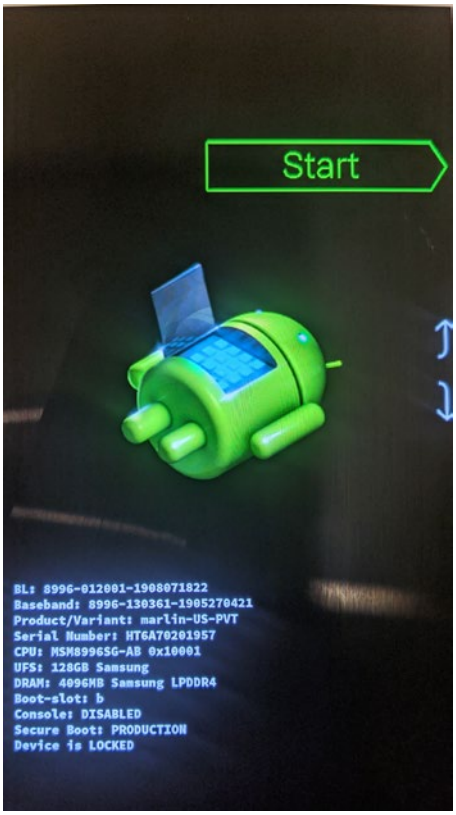


Figure 8-9. *The Android bootloader screen*

You will notice that the last line says Device is LOCKED. Let's unlock it. With your device still connected, go to your workstation command line and issue the command (remember issuing this command wipes all your data):

```
➜  fastboot flashing unlock
                                                    OKAY [  0.034s]
Finished. Total time: 0.035s
➜
```

On your device, you should then see a prompt asking you to confirm. This screen is shown in Figure 8-10. Use the volume up and down keys to select the Yes option and then press the power button to confirm.

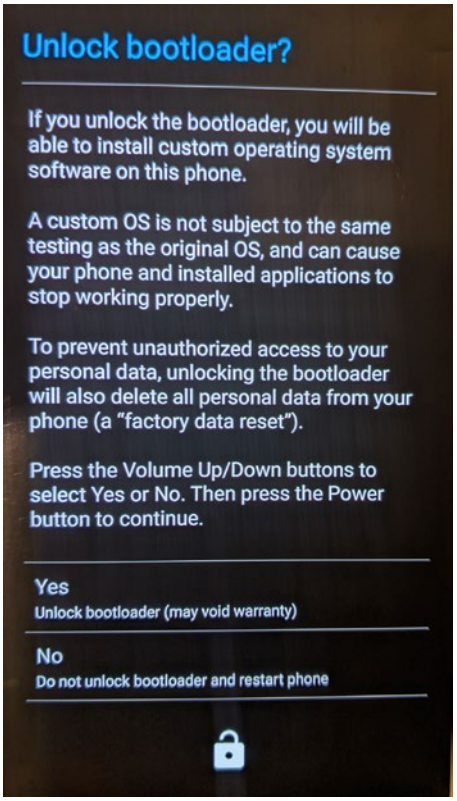


Figure 8-10. *Confirmation on whether to unlock the bootloader*

Flashing the Modified boot.img

After this, the device goes back to the bootloader screen, and now the screen should say Device is UNLOCKED. At this time, we are ready to flash our Magisk patched `boot.img`. Go to the directory where you downloaded the `magisk_patched.img` file previously, and then issue the command:

```
➜  fastboot flash boot magisk_patched.img
Sending 'boot_b' (31189 KB)                        OKAY [  0.851s]
Writing 'boot_b'                                   OKAY [  0.297s]
Finished. Total time: 1.301s
➜
```

This should flash the new `boot.img` onto the device. At this point, either the phone will restart by itself or you will have to restart it. On the bootloader screen, press the

volume up or down keys until you see the green arrow with the word Start like in Figure 8-9. Then press the power button to restart the device. You will see now that the startup screens have been altered somewhat. First, in Figure 8-11, you will see this warning telling you that the bootloader has been unlocked and that Android cannot check the integrity of your device software. Then, you will notice in Figure 8-12 an unlocked padlock icon on the Google startup screen.

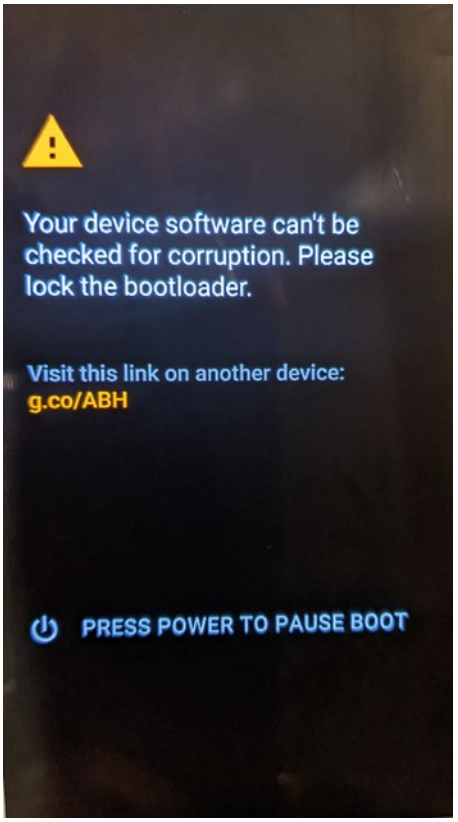


Figure 8-11. *The warning that the bootloader has been unlocked*

Figure 8-12. *The Google startup screen with unlocked padlock icon*

Completing the Rooting Process

After this, allow your device to reboot and then configure it as you need to by adding your Wi-Fi, Google account, and others. Also you need to reenable Developer Options and turn on USB Debugging. Then, reinstall Magisk Manager like before and start it up. Magisk should now prompt you to complete your setup. Click OK and make sure that you're connected to the Internet. Magisk will continue the installation and then automatically reboot once again. When the reboot completes, open Magisk once again, and you should now see that Magisk Manager says Magisk is installed (Figure 8-13).

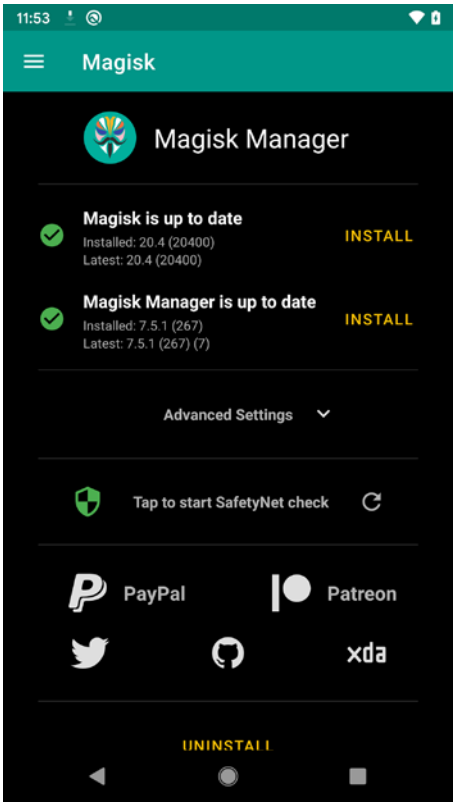


Figure 8-13. *Magisk now shows up as installed*

Let's test out root now. On `adb` open up an `adb` shell and then see if you are able to `su` to the root user:

```
➜  adb shell
marlin:/ $ su
marlin:/ #
```

When you switch to root by typing su, you should receive a prompt on your device from Magisk that looks like Figure 8-14. Pick for how long you want to grant root privileges and then click the Grant button.

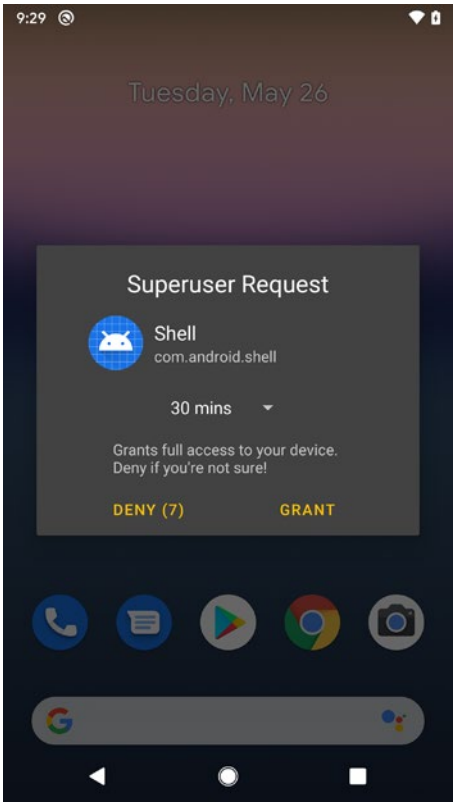


Figure 8-14. *The Magisk superuser permission request window*

You will notice that Magisk tells you that if you're not sure, you should deny permission. In case you were rooted without your knowledge using Magisk, then you will get this notification each time a program wants root access. It's kind of a safety mechanism. That's it, you now have root on your device!

Looking a Little Bit Deeper

So what *does* Magisk actually do under the hood? I took a closer look at what Magisk does for the Google Pixel, and essentially, it boils down to a few main things.

Magisk will patch the ramdisk and copy over the magiskinit binary to the ramdisk. It will extract the magisk binary from within the magiskinit file. It then configures the boot process of the device so that it runs itself first, does some setup tasks required for staying active as a service, and then calls the normal boot process.

Magisk will patch the kernel at offset 0x017853f6 to require the loading of the ramdisk from the kernel (Figure 8-15 shows the Magisk patch when applied to the kernel). This step is essential so that the patches to the ramdisk can take effect.

```
.text:00000000017853D9 __setup_str_prompt_ramdisk DCB "prompt_ramdisk=",0
.text:00000000017853E9 __setup_str_no_initrd DCB "noinitrd",0
.text:00000000017853F2 __setup_str_dm_setup DCB "dm=",0
.text:00000000017853F6 __setup_str_skip_initramfs_param DCB "want_initramfs",0
.text:0000000001785405 __setup_str_retain_initrd_param DCB "retain_initrd",0
.text:0000000001785413 __setup_str_lpj_setup DCB "lpj=",0
.text:0000000001785418 __setup_str_early_debug_disable DCB "nodebugmon",0
```

Figure 8-15. *The patch that Magisk made from skip_initramfs to want_initramfs*

Further details can be found on the Magisk website (https://topjohnwu.github.io/Magisk/details.html and https://topjohnwu.github.io/Magisk/tools.html).

Other Ways of Rooting

If we are purely after rooting for the purposes of being able to debug our applications and not so much for customizing the look and feel of the device, then it is possible to build an Android image from scratch using the AOSP source code. I state it here as an option, but will not be going into depth in this book.

Testing Frida

Now that we have root on our device, let's do a test with Frida but this time using our device instead of the emulator. There is one change that you will have to make and that is in uploading the server. When we used Frida on our emulator, we had to download the Frida x86 server. For my Google Pixel XL, I will use the Frida ARM64 server. To get this, head down to the Frida releases [https://github.com/frida/frida/releases] page and pick up the server that says `frida-server-12.9.4-android-arm64.xz`. Then, as before, you will have to decompress the image using `unxz` and push it to your device:

```
(p3) ➜  unxz frida-server-12.9.4-android-arm64.xz
(p3) ➜  adb push frida-server-12.9.4-android-arm64 /data/local/tmp/
        frida-server
```

Then get a shell into your device, switch to the root user, and run the server:

```
1: (p3) ➜  adb shell
2: marlin:/ $ su
3: marlin:/ # cd /data/local/tmp
4: marlin:/data/local/tmp # chmod u+x frida-server
5: marlin:/data/local/tmp # chown root:root frida-server
6: marlin:/data/local/tmp # ./frida-server &
```

Now that we have the Frida server running, let's try to do a simple experiment with breaking SSL Pinning. I have tested and know that the Google Play Store and pretty much all Google apps will use SSL Pinning. Let's try to bypass that so that we can take a look at what traffic goes back and forth between the device and the Play Store whenever we're browsing the store. First, get Burp Proxy up and running and start the proxy on port 8888 by going to Proxy ➤ Options ➤ Proxy Listeners ➤ Add. Enter your port and host IP address for your workstation running Burp. Next, configure your proxy to point to the host running Burp. Figure 8-16 shows a screenshot of my Pixel XL.

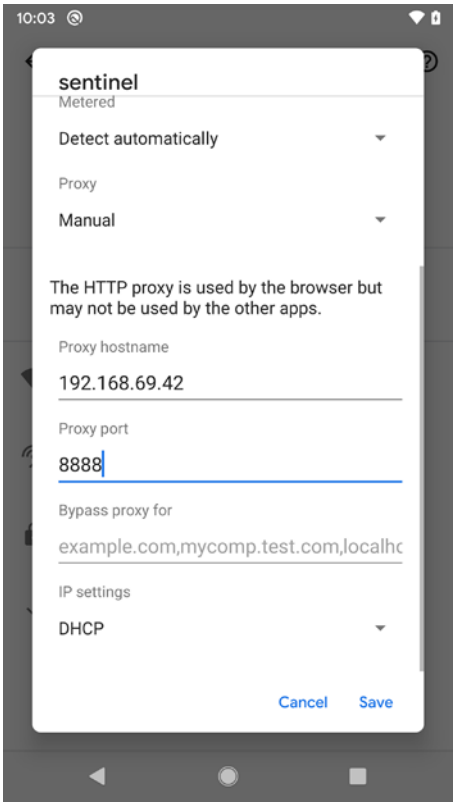


Figure 8-16. *Configuring the proxy for your device's Wi-Fi connection*

We will use the frida-multiple-unpinning script to defeat SSL Pinning of the Google Play Store. On your workstation or host, run the following:

```
(p3) ➜  frida -U --codeshare akabe1/frida-multiple-unpinning -f
        com.android.vending --no-pause

     ____
    / _  |   Frida 12.9.4 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://www.frida.re/docs/home/
Spawned `com.android.vending`. Resuming main thread!
[Pixel XL::com.android.vending]->
======
[#] Android Bypass for various Certificate Pinning methods [#]
======
[-] OkHTTPv3 pinner not found
[-] Trustkit pinner not found
[-] Appcelerator PinningTrustManager pinner not found
[-] OpenSSLSocketImpl Conscrypt pinner not found
[-] OpenSSLEngineSocketImpl Conscrypt pinner not found
[-] OpenSSLSocketImpl Apache Harmony pinner not found
[-] PhoneGap sslCertificateChecker pinner not found
[-] IBM MobileFirst pinTrustedCertificatePublicKey pinner not found
[-] IBM WorkLight HostNameVerifierWithCertificatePinning pinner not found
[-] Conscrypt CertPinManager pinner not found
[-] CWAC-Netsecurity CertPinManager pinner not found
[-] Worklight Androidgap WLCertificatePinningPlugin pinner not found
[-] Netty FingerprintTrustManagerFactory pinner not found
[-] Squareup CertificatePinner pinner not found
[-] Squareup OkHostnameVerifier pinner not found
[-] Apache Cordova WebViewClient pinner not found
[-] Boye AbstractVerifier pinner not found
```

```
[+] Bypassing TrustManagerImpl (Android > 7): connectivitycheck.gstatic.com
[+] Bypassing TrustManagerImpl (Android > 7): android.clients.google.com
[+] Bypassing TrustManagerImpl (Android > 7): lh3.googleusercontent.com
```

In the last three lines, you can see that the script has already detected SSL Pinning using the `TrustManager` and has bypassed it. To see all the requests, you will have to change the filter on Burp Proxy. Under the Proxy ➤ HTTP history tab, click the section that says "Filter:..." as shown in Figure 8-17. Then click the Show all button and close the window.

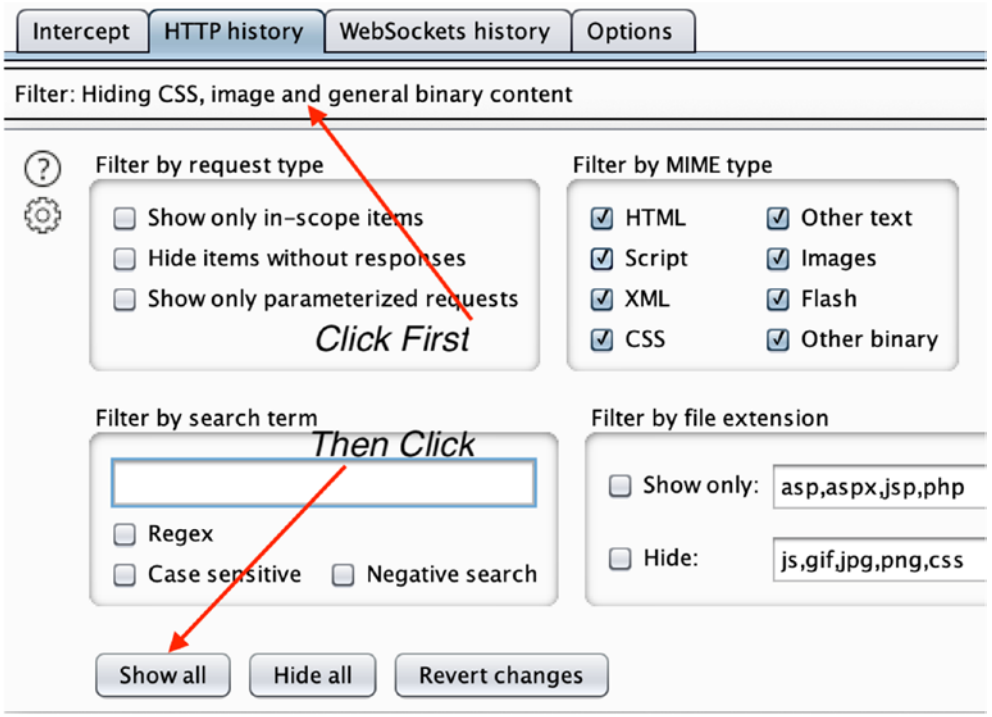


Figure 8-17. *Changing Burp Proxy's filter settings to show all traffic*

Then select some apps on your Google Play Store on your device and take a look at your Burp HTTP history tab. You should now be able to see the requests and responses going back and forth between the device and the server as shown in Figures 8-18 and 8-19, respectively.

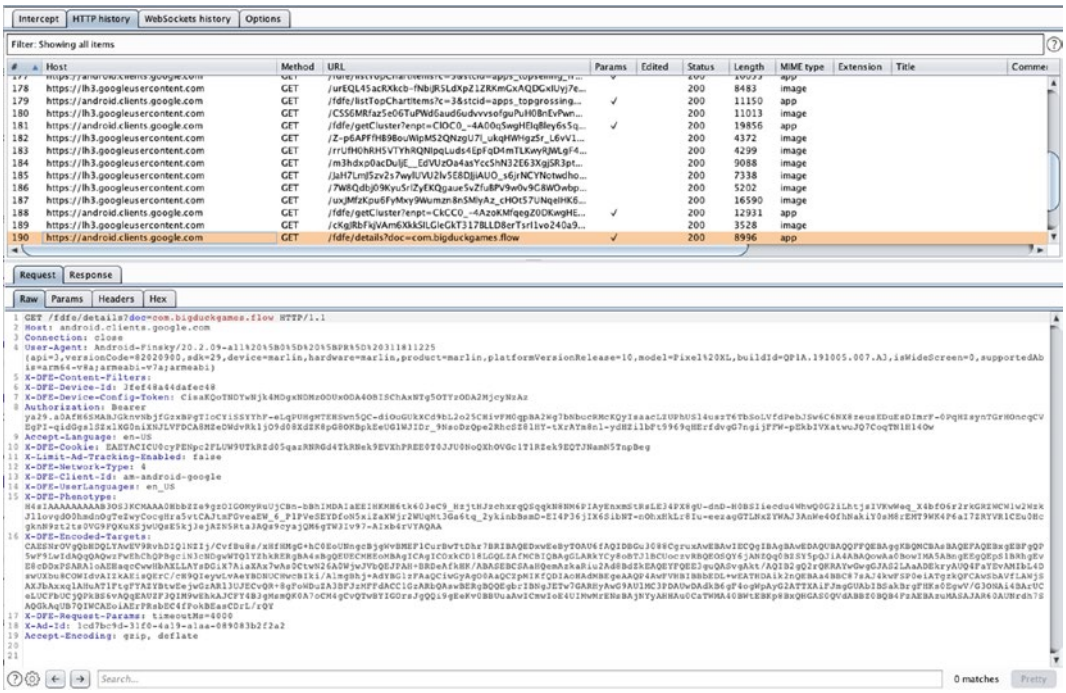


Figure 8-18. *HTTPS traffic requests going from the device and Google Play Store*

Figure 8-19. *HTTPS traffic responses returned from Google Play Store*

Examining the Filesystem

One of the perks of having root is unfettered access throughout the Android filesystem. I want to spend this section diving into the /data partition. The /system partition contains the operating system and most of that remains static, and for our purposes of debugging or reverse engineering, we don't necessarily need to look there. The /data partition, on the other hand, is where all user and app data reside. Let's look at two of our apps: aas2client and aas2obfuscate. Here are their directory layouts:

```
com.redteamlife.aas2.aas2client/
|-- cache
|-- code_cache
|-- databases
|-- files
|-- no_backup
`-- shared_prefs

com.redteamlife.aas2.aas2obfuscate/
|-- cache
|-- code_cache
|-- databases
|-- files
|-- no_backup
`-- shared_prefs
```

The app data is stored in the `/data/user/0` directory, and then underneath this directory, there are the subdirectories that correspond to each app. Here's a snippet:

```
marlin:/data/user/0 # ls -al |tail
drwx------   8 u0_a187        u0_a187         4096 2020-05-25 23:47
com.redteamlife.aas2.aas2obfuscate
drwx------   8 u0_a174        u0_a174         4096 2020-05-16 23:50
com.topjohnwu.magisk
drwx------   4 u0_a89         u0_a89          4096 2019-10-29 07:37
com.ustwo.lwp
drwx------   4 u0_a61         u0_a61          4096 2019-10-29 07:37
com.verizon.llkagent
```

```
drwx------   8 u0_a118        u0_a118         4096 2020-05-16 23:46
com.verizon.obdm
drwx------   4 u0_a120        u0_a120         4096 2019-10-29 07:37
com.verizon.obdm_permissions
drwx------   8 u0_a55         u0_a55          4096 2020-05-16 23:46
com.verizon.services
drwx------   8 u0_a145        u0_a145         4096 2020-05-16 23:46
com.vzw.apnlib
drwx------   4 radio          radio           4096 2019-10-29 07:37
org.codeaurora.ims
drwx------   8 u0_a92         u0_a92          4096 2020-05-16 23:46
qualcomm.com.vzw_msdc_api
marlin:/data/user/0 #
```

Notice how each directory is owned by a different user with no permissions being assigned to any other users. I took a look inside our app directories, and since they are such simple apps, none of the directories in each app data directory is populated. As a better example, let's look at Google Chrome. Here's what the Chrome directory structure looks like:

```
com.android.chrome/
|-- app_chrome
|   |-- BrowserMetrics
|   |-- BrowserMetrics-spare.pma
|   |-- CertificateRevocation
|   |-- Crowd\ Deny
|   |-- Default
|   |-- Default.previews_hint_cache_store
|   |-- FileTypePolicies
|   |-- GrShaderCache
|   |-- Local\ State
|   |-- OnDeviceHeadSuggestModel
|   |-- OriginTrials
|   |-- SSLErrorAssistant
|   |-- SafetyTips
|   |-- ShaderCache
|   |-- Subresource\ Filter
|   `-- TLSDeprecationConfig
```

```
|-- app_dex
|   |-- oat
|   `-- webapk6.dex
|-- app_tabs
|   `-- 0
|-- app_textures
|   `-- 0
|-- cache
|   |-- Cache
|   |-- Code\ Cache
|   |-- Crash\ Reports
|   |-- Crashpad
|   |-- Offline\ Pages
|   |-- font_unique_name_table.pb
|   `-- image_cache
|-- code_cache
|-- databases
|-- files
|   |-- images
|   `-- splitcompat
|-- lib -> /data/app/com.android.chrome-uekNic08OMP4vIsHfP_jag==/lib/arm
|-- no_backup
|   |-- com.google.InstanceId_Y29tLmdvb2dsZS5jaHJvbWUuT2ZmbGluZVBhZ2VQcmVmZ
        XRjaA.properties
|   `-- com.google.android.gms.appid-no-backup
`-- shared_prefs
    |-- com.android.chrome_preferences.xml
    |-- com.google.android.apps.chrome.omaha.xml
    |-- com.google.android.gms.appid.xml
    |-- com.google.android.libraries.hats20.xml
    |-- icing_firebase_pref.xml
    |-- org.chromium.components.background_task_scheduler.xml
    |-- org.chromium.components.gcm_driver.subscription_flags.xml
    `-- webapp_registry.xml
```

Now you will notice more directories in the Chrome directory (I'm only going two levels down). Let's first talk about the main directories created – starting with the cache directory. As the name implies, this directory holds the app's cached set of files. In this example, it holds files like offline files, images, fonts, and crash reports. For example, if we go into the `image_cache` directory, we see that it stores some image files in PNG format:

```
marlin:/data/user/0/com.android.chrome/cache # cd image_cache/image_data_
storage/
marlin:/data/user/0/com.android.chrome/cache/image_cache/image_data_storage
# ls -al
total 3052
drwx--S--- 2 u0_a138 u0_a138_cache    4096 2020-05-26 15:58 .
drwx--S--- 4 u0_a138 u0_a138_cache    4096 2020-05-26 15:21 ..
-rw------- 1 u0_a138 u0_a138_cache  165638 2020-05-26 15:58
3BMOCQPJKWUIAPFKUCBRFWKGDFGOQKKG
-rw------- 1 u0_a138 u0_a138_cache  184090 2020-05-26 15:58
4WPXVZNDEHD7GETEAWWHTRYJVQYCMXZ2
-rw------- 1 u0_a138 u0_a138_cache    1438 2020-05-26 15:58
FCHAZF57RCYRNFMRXPZSLFMQIURR2PM7
-rw------- 1 u0_a138 u0_a138_cache     746 2020-05-26 15:58
FUY2CBED2WAUJDJHG4TFNOFDMEYQBI5U
-rw------- 1 u0_a138 u0_a138_cache  101556 2020-05-26 15:58
GMKJWLRURBIRM3RS5SMLJCAQI3D6GBHI
-rw------- 1 u0_a138 u0_a138_cache  112219 2020-05-26 15:58
HXCYNXFALMMGEEYSYGYKQAQYWZQTTH53
-rw------- 1 u0_a138 u0_a138_cache 1821170 2020-05-26 15:58
L345GKEPDJNJKHMFXHIPHGM7Q2IKIZW2
-rw------- 1 u0_a138 u0_a138_cache  131643 2020-05-26 15:58
LXHJ376LY7HJWCMKSYPANTXBY7DB3WQG
-rw------- 1 u0_a138 u0_a138_cache    2281 2020-05-26 15:58
M5GHVE2O64LCK2KD4RYPBTR3DZ2ZQKIR
-rw------- 1 u0_a138 u0_a138_cache  175446 2020-05-26 15:21
RVPMIXPZFAH3GITNBPSCZ6T3JXOXMGYH
-rw------- 1 u0_a138 u0_a138_cache    2703 2020-05-26 15:58
SSEZ7WZFKL6VBFQMCJ6UA3PMOHGB75WW
```

```
-rw------- 1 u0_a138 u0_a138_cache  192894 2020-05-26 15:58
UF34JJ7XOK6AXTMTARBMWZRG353H4G4H
-rw------- 1 u0_a138 u0_a138_cache    2344 2020-05-26 15:21
WE4GEZR5PCHUO5N3KB3RKCLJP4AD2BGO
-rw------- 1 u0_a138 u0_a138_cache    2883 2020-05-26 15:58
XC25K6GUFO5KCCOLBVGG6RBHFPUGUVAB
-rw------- 1 u0_a138 u0_a138_cache  119730 2020-05-26 15:58
XL4QHG75GDRZHFXBH4DH4N5FEVS6KLY3
marlin:/data/user/0/com.android.chrome/cache/image_cache/image_data_storage
# file XL4QHG75GDRZHFXBH4DH4N5FEVS6KLY3
XL4QHG75GDRZHFXBH4DH4N5FEVS6KLY3: PNG image data, 322 x 322, 8-bit/color
RGBA, non-interlaced
marlin:/data/user/0/com.android.chrome/cache/image_cache/image_data_storage #
```

When you select "Clear Cache" on your App Info screen, this is the directory that it clears. Generally, the data in this directory can be cleared as it is transient and is used to speed things up by either prefetching data or storing frequently accessed data. Chrome stores its open tabs and the state of those tabs in the directory called app_tabs. Inside this directory, you will see the entries for all the tabs. If you look at my tabs, you can see that I have one tab open that is pointing to Wikipedia. The structure of each file is not something we will go into, but by running the strings command on the file, you can see what printable strings there are. Based on those results, you can quickly determine what the tab contents are.

```
2|marlin:/data/user/0/com.android.chrome # cd app_tabs/0
marlin:/data/user/0/com.android.chrome/app_tabs/0 # ls -al
total 32
drwx------ 2 u0_a138 u0_a138 4096 2020-05-26 16:33 .
drwxrwx--x 3 u0_a138 u0_a138 4096 2020-05-26 15:21 ..
-rw------- 1 u0_a138 u0_a138 1771 2020-05-26 16:48 tab0
-rw------- 1 u0_a138 u0_a138   67 2020-05-26 16:33 tab_state0
127|marlin:/data/user/0/com.android.chrome/app_tabs/0 # strings tab0
rP+s7
chrome-native://newtab/
chrome-native://newtab/
https://en.m.wikipedia.org/wiki/Main_Page
https://en.m.wikipedia.org/
marlin:/data/user/0/com.android.chrome/app_tabs/0 #
```

A note about the no_backup directory: You will notice this directory inside all app data directories. Files that are placed in this directory do not get backed up. During an app or full backup of data, these files will be skipped over and ignored. Therefore, if there is sensitive data that you don't want to go into a user's backup, then you would place those files here. In Kotlin, you access the directory by using Context.getNoBackupDir [https://developer.android.com/reference/android/content/Context#getNoBackupFilesDir()] like this:

```
val noBackupDir = applicationContext.noBackupFilesDir
```

Then you can write or read files from this directory as you would normally using getFilesDir() [https://developer.android.com/reference/android/content/Context#getFilesDir()].

The databases directory typically contains your structured data. You can do this by using SQLite or by using Room [https://developer.android.com/training/data-storage/room] which is an abstraction layer that takes care of reading and writing your data to SQLite databases. Google recommends that you use Room instead of SQLite but doesn't limit your use of SQLite. Let's look at a concrete example of data in the databases directory. Remember our old friend the NYTimes – Crossword app? Let's take a look at the databases directory in that app.

```
marlin:/ # cd /data/user/0/com.nytimes.crossword/databases
marlin:/data/user/0/com.nytimes.crossword/databases # ls -al
total 808
drwxrwx--x  2 u0_a180 u0_a180   4096 2020-05-27 13:02 .
drwx------ 11 u0_a180 u0_a180   4096 2020-05-27 13:02 ..
-rw-rw----  1 u0_a180 u0_a180 458752 2020-05-27 13:02 Crosswords.db
-rw-rw----  1 u0_a180 u0_a180      0 2020-05-27 13:02 Crosswords.db-journal
-rw-rw----  1 u0_a180 u0_a180   4096 2020-05-27 13:02 androidx.work.workdb
-rw-------  1 u0_a180 u0_a180  32768 2020-05-27 13:02 androidx.work.workdb-shm
-rw-------  1 u0_a180 u0_a180  90672 2020-05-27 13:02 androidx.work.workdb-wal
-rw-rw----  1 u0_a180 u0_a180  24576 2020-05-27 13:02 com.microsoft.
appcenter.persistence
```

```
-rw-rw----  1 u0_a180 u0_a180      0 2020-05-27 13:02 com.microsoft.
appcenter.persistence-journal
-rw-rw----  1 u0_a180 u0_a180   4096 2020-05-27 13:02 event-buffer.db
-rw-------  1 u0_a180 u0_a180  32768 2020-05-27 13:02 event-buffer.db-shm
-rw-------  1 u0_a180 u0_a180  86552 2020-05-27 13:02 event-buffer.db-wal
-rw-rw----  1 u0_a180 u0_a180  16384 2020-05-27 13:02 google_app_
measurement_local.db
-rw-rw----  1 u0_a180 u0_a180      0 2020-05-27 13:02 google_app_
measurement_local.db-journal
marlin:/data/user/0/com.nytimes.crossword/databases #
```

Hmm, that `Crosswords.db` looks interesting. Let's take a look at that. Some device manufacturers will include the sqlite3 binary in your device. Some do not. My Google Pixel doesn't come with it so I will copy the database file to my Mac and work on it there. I will first copy it over to `/data/local/tmp` and then use `adb pull` to download it to my Mac:

On the device

```
marlin:/data/user/0/com.nytimes.crossword/databases # ls -al
total 808
drwxrwx--x  2 u0_a180 u0_a180   4096 2020-05-27 13:02 .
drwx------ 11 u0_a180 u0_a180   4096 2020-05-27 13:02 ..
-rw-rw----  1 u0_a180 u0_a180 458752 2020-05-27 13:02 Crosswords.db
-rw-rw----  1 u0_a180 u0_a180      0 2020-05-27 13:02 Crosswords.db-journal
-rw-rw----  1 u0_a180 u0_a180   4096 2020-05-27 13:02 androidx.work.workdb
-rw-------  1 u0_a180 u0_a180  32768 2020-05-27 13:02 androidx.work.workdb-shm
-rw-------  1 u0_a180 u0_a180  90672 2020-05-27 13:02 androidx.work.workdb-wal
-rw-rw----  1 u0_a180 u0_a180  24576 2020-05-27 13:02 com.microsoft.
appcenter.persistence
-rw-rw----  1 u0_a180 u0_a180      0 2020-05-27 13:02 com.microsoft.
appcenter.persistence-journal
-rw-rw----  1 u0_a180 u0_a180   4096 2020-05-27 13:02 event-buffer.db
-rw-------  1 u0_a180 u0_a180  32768 2020-05-27 13:02 event-buffer.db-shm
-rw-------  1 u0_a180 u0_a180  86552 2020-05-27 13:02 event-buffer.db-wal
-rw-rw----  1 u0_a180 u0_a180  16384 2020-05-27 13:02 google_app_
                                                      measurement_local.db
```

```
-rw-rw----  1 u0_a180 u0_a180      0 2020-05-27 13:02 google_app_
measurement_local.db-journal
marlin:/data/user/0/com.nytimes.crossword/databases # cp Crosswords.db
/data/local/tmp
marlin:/data/user/0/com.nytimes.crossword/databases # chown shell:shell
/data/local/tmp/Crosswords.db
```

On the Mac

```
➜  adb pull /data/local/tmp/Crosswords.db
/data/local/tmp/Crosswords.db: 1 file pulled, 0 skipped. 13.7 MB/s
(458752 bytes in 0.032s)
➜
```

Now that the file is on my Mac, we can open it in sqlite3 and look at its schema:

```
➜  sqlite3 Crosswords.db
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE `GameData`(`puzzleId` INTEGER,`puzzlePackIap`
TEXT,`formatType` TEXT,`publishType` TEXT,`printDate` TEXT,`jsonData`
TEXT,`author` TEXT,`editor` TEXT,`title` TEXT, PRIMARY KEY(`puzzleId`));
CREATE TABLE `CommitLog`(`id` INTEGER,`puzzleId` INTEGER,`committed`
INTEGER,`commitId` TEXT,`resetTimer` INTEGER,`timestamp` INTEGER,`board`
TEXT,`timerDiff` INTEGER, PRIMARY KEY(`id`));
CREATE TABLE `GameState`(`puzzleId` INTEGER,`status` TEXT,`firstOpened`
INTEGER,`firstSolved` INTEGER,`firstCleared` INTEGER,`firstRevealed`
INTEGER,`firstTimerReset` INTEGER,`timeSpentInPuzzle`
INTEGER,`percentComplete` INTEGER,`lastUpdateTime` INTEGER,`solved`
INTEGER, PRIMARY KEY(`puzzleId`));
CREATE TABLE `Streak`(`id` INTEGER,`startDate` INTEGER,`endDate` INTEGER,
PRIMARY KEY(`id`));
CREATE TABLE `LegacyGameProgress`(`id` INTEGER,`puzzleId` INTEGER,`value`
TEXT,`timestamp` INTEGER,`cellIndex` INTEGER,`mode` INTEGER, PRIMARY
KEY(`id`));
```

```
CREATE TABLE `GamesHubData`(`id` INTEGER,`puzzleId` INTEGER,`title`
TEXT,`author` TEXT,`editor` TEXT,`formatType` TEXT,`publishType`
TEXT,`printDate` TEXT, PRIMARY KEY(`id`));
CREATE TABLE `PuzzlePack`(`productId` INTEGER,`platform`
TEXT,`iapProductId` TEXT,`productType` TEXT,`createdDate`
TEXT,`lastModifiedTimestamp` TEXT,`liveStartDateTimestamp`
TEXT,`liveEndDateTimestamp` TEXT,`packName` TEXT,`status` TEXT,`iconUrl`
TEXT,`numberOfPuzzlesInPack` INTEGER, PRIMARY KEY(`productId`));
CREATE TABLE `GoldStarDay`(`id` INTEGER,`date` INTEGER,`printDate`
TEXT,`dayOffset` INTEGER,`dayOfWeek` INTEGER, PRIMARY KEY(`id`));
CREATE TABLE `GameProgress`(`id` INTEGER,`puzzleId` INTEGER,`value`
TEXT,`timestamp` INTEGER,`cellIndex` INTEGER,`mode`
INTEGER,`puzzleOpenedTimestamp` INTEGER,`puzzleSolvedTimestamp` INTEGER,
PRIMARY KEY(`id`));
CREATE TABLE `PuzzleForPack`(`packID` INTEGER,`puzzleID` INTEGER, PRIMARY
KEY(`packID`,`puzzleID`));
sqlite>
```

Look at the GameData table. It has a field called jsonData; let's take a look at that:

```
sqlite> select jsonData from GameData;
...
...
... Rows of other data
...
...
{"status":"OK","entitlement":"premium","results":[{"puzzle_id":18293,
"version":0,"puzzle_meta":{"formatType":"Normal","publishType":"Mini",
"title":"","printDate":"2020-05-27","printDotw":3,"editor":"","copyright":
"2020, The New York Times","height":5,"width":5,"links":[],"layoutExtra":[],
"author":"Joel Fagliano","notes":[]},"puzzle_data":{"answers":[null,"H",
"O","T",null,"N","O","B","E","L","F","R","A","M","E","L","U","M","P","S",
null,"S","A","T",null],"clues":{"A":[{"formatted":"Like summer, or a word
before \u0026#34;spring\u0026#34;","squares":[],"clueNum":"1","clueStart":1,
"value":"Like summer, or a word before \"spring\"","clueEnd":3,"index":0},
{"squares":[],"clueNum":"4","clueStart":5,"value":"Prize for Malala
```

Yousafzai","clueEnd":9,"index":0},{"squares":[],"clueNum":"6","clueStart":10,
"value":"Falsely pin for a crime","clueEnd":14,"index":0},{"squares":[],
"clueNum":"7","clueStart":15,"value":"Problems with mashed potatoes",
"clueEnd":19,"index":0},{"squares":[],"clueNum":"8","clueStart":21,"value":
"Test no longer required for University of California admission",
"clueEnd":23,"index":0}],"D":[{"squares":[],"clueNum":"1","clueStart":1,
"value":"Egyptian god with the head of a falcon","clueEnd":21,"index":0},
{"formatted":"Subject of the 2020 documentary \u0026#34;Becoming\u0026#34;",
"squares":[],"clueNum":"2","clueStart":2,"value":"Subject of the 2020
documentary \"Becoming\"","clueEnd":22,"index":0},{"squares":[],"clueNum":
"3","clueStart":3,"value":"Entice","clueEnd":23,"index":0},{"squares":[],
"clueNum":"4","clueStart":5,"value":"Org. for Falcons, Eagles and Ravens",
"clueEnd":15,"index":0},{"formatted":"\"Allez ___ Bleus!\" (French soccer
cheer)","squares":[],"clueNum":"5","clueStart":9,"value":"\"Allez ___
Bleus!\" (French soccer cheer)","clueEnd":19,"index":0}]},"layout":
[0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0]}}]}
sqlite>

Look at that! All our clues and answers are stored in this database. In our previous hacking chapter, we pulled this data out the hard way even though I knew all along that you could pull it from the database, but the point I was trying to make is that there are other ways in which you can fetch data that an app uses.

The `shared_prefs` directory will usually be the place that apps store their key-value data. The way Android writes these preferences to the filesystem is via XML files. Again, using our same Crossword app, the `shared_prefs` looks like this:

```
marlin:/data/user/0/com.nytimes.crossword # cd shared_prefs/
marlin:/data/user/0/com.nytimes.crossword/shared_prefs # ls -al
total 200
drwxrwx--x  2 u0_a180 u0_a180 4096 2020-05-27 13:02 .
drwx------ 11 u0_a180 u0_a180 4096 2020-05-27 13:02 ..
-rw-rw----  1 u0_a180 u0_a180  490 2020-05-27 13:02 AppCenter.xml
-rw-rw----  1 u0_a180 u0_a180  436 2020-05-27 13:02
EntitlementsAndPurchase.xml
```

```
-rw-rw----  1 u0_a180 u0_a180  240 2020-05-27 13:02 NYTIMES_PREFS.xml
-rw-rw----  1 u0_a180 u0_a180  213 2020-05-27 13:02
TwitterAdvertisingInfoPreferences.xml
-rw-rw----  1 u0_a180 u0_a180  127 2020-05-27 13:02 WebViewChromiumPrefs.xml
-rw-rw----  1 u0_a180 u0_a180  862 2020-05-27 13:02 appsflyer-data.xml
-rw-rw----  1 u0_a180 u0_a180  251 2020-05-27 13:02 com.crashlytics.prefs.xml
-rw-rw----  1 u0_a180 u0_a180  125 2020-05-27 13:02 com.crashlytics.sdk.
android:answers:settings.xml
-rw-rw----  1 u0_a180 u0_a180   65 2020-05-27 13:02 com.facebook.
AccessTokenManager.SharedPreferences.xml
-rw-rw----  1 u0_a180 u0_a180  127 2020-05-27 13:02 com.facebook.internal.
SKU_DETAILS.xml
-rw-rw----  1 u0_a180 u0_a180 2042 2020-05-27 13:02 com.facebook.internal.
preferences.APP_GATEKEEPERS.xml
-rw-rw----  1 u0_a180 u0_a180 1453 2020-05-27 13:02 com.facebook.internal.
preferences.APP_SETTINGS.xml
-rw-rw----  1 u0_a180 u0_a180  129 2020-05-27 13:02 com.facebook.
loginManager.xml
-rw-rw----  1 u0_a180 u0_a180  138 2020-05-27 13:02 com.facebook.sdk.USER_
SETTINGS.xml
-rw-rw----  1 u0_a180 u0_a180  160 2020-05-27 13:02 com.facebook.sdk.
appEventPreferences.xml
-rw-rw----  1 u0_a180 u0_a180 2512 2020-05-27 13:02 com.google.android.gms.
appid.xml
-rw-rw----  1 u0_a180 u0_a180  999 2020-05-27 13:02 com.google.android.gms.
measurement.prefs.xml
-rw-rw----  1 u0_a180 u0_a180  127 2020-05-27 13:02 com.google.firebase.
remoteconfig_legacy_settings.xml
-rw-rw----  1 u0_a180 u0_a180  192 2020-05-27 13:02 com.mobileapptracking.xml
-rw-rw----  1 u0_a180 u0_a180  339 2020-05-27 13:02 com.nytimes.android.
eventtracker.CLOCK_CACHE.xml
-rw-rw----  1 u0_a180 u0_a180 2848 2020-05-27 13:02 com.nytimes.crossword_
preferences.xml
```

```
-rw-rw----  1 u0_a180 u0_a180  181 2020-05-27 13:02 com.tune.ma.profile.xml
-rw-rw----  1 u0_a180 u0_a180  381 2020-05-27 13:02 frc_1:945293061443:andr
oid:9db0c9c54c4f0fad_firebase_settings.xml
marlin:/data/user/0/com.nytimes.crossword/shared_prefs #
```

Let's dump the contents of `com.nytimes.crossword_preferences.xml` to see what's in there:

```
marlin:/data/user/0/com.nytimes.crossword/shared_prefs # cat com.nytimes.
crossword_preferences.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <boolean name="HAPTIC_ENABLED" value="true" />
    <long name="LAST_ONBOARDING_TIMESTAMP" value="1590555743495" />
    <boolean name="JUMP_TO_NEXT_CLUE" value="true" />
    <int name="LAST_PUSH_REGISTERED_VERSION" value="668" />
    <boolean name="DAILY_MINI_SWITCH_PREF" value="true" />
    <boolean name="HAS_WRITTEN_DEFAULTS" value="true" />
    <boolean name="SHOW_TIMER" value="true" />
    <boolean name="SHOW_OVERLAY" value="true" />
    <long name="PUZZLE_REFRESH_TIMESTAMP" value="1590555758807" />
    <string name="kst">trial</string>
    <boolean name="DARK_MODE_KEY" value="false" />
    <long name="GDPR_LAST_TS" value="1590555744112" />
    <long name="com.nytimes.android.eventtracker.KEY_SESSION_INDEX"
    value="1" />
    <set name="SUBSCRIBED_PUSH_TAGS" />
    <int name="12881-SELECTED_CELL_KEY" value="1" />
    <string name="MINI_LAST_SELECTED_MONTH_PREF">May 2020</string>
    <string name="12881-CURRENT_DIRECTION_STR">Across</string>
    <boolean name="com.nytimes.android.eventtracker.KEY_SESSION_TIME_TYPE"
    value="false" />
    <boolean name="LIGHT_MODE_KEY" value="true" />
    <string name="THEME_MODE">LIGHT</string>
    <string name="Purr.Directives">{&amp;quot;adConfiguration&amp;quot;:{&amp;quot;
    value&amp;quot;:&amp;quot;FULL&amp;quot;},&amp;quot;acceptableTrackers&amp;quot;:{&amp;quot;
```

```
    value&quot;:&quot;CONTROLLERS&quot;},&quot;showDataSaleOptOutDirective
    &quot;:{&quot;show&quot;:false,&quot;preference&quot;:&quot;NYT_SELL_
    PERSONAL_INFORMATION_CCPA&quot;}}</string>
    <string name="Purr.Directives.LastTS">2020-05-27T05:02:24.361Z</string>
    <boolean name="PROGRESS_MILESTONES_DEFAULT" value="true" />
    <boolean name="SKIP_FILLED_PENCILLED" value="false" />
    <string name="SmartLockTask.KEY_LAST_CHECK">3.0.1</string>
    <string name="DeviceID">17dbb2ee01d6ab06cf1237331a9c162f</string>
    <boolean name="ENTITLED_TO_DAILY_IN_TRIAL_PREF" value="true" />
    <string name="ktr">9319a3ea0c48e16e</string>
    <long name="com.nytimes.android.eventtracker.KEY_LAST_EVENT_TIME_VALUE"
    value="1590555758999" />
    <boolean name="SHOULD_PROMO_NEW_SETTING" value="false" />
    <boolean name="JUMP_BACK" value="true" />
    <boolean name="IS_GDPR" value="false" />
    <int name="CURRENT_STREAK" value="0" />
    <boolean name="LEADERBOARD_UPGRADED" value="true" />
    <long name="ktrts" value="1591117342000" />
    <boolean name="com.nytimes.android.eventtracker.KEY_LAST_EVENT_TIME_
    TYPE" value="false" />
    <long name="com.nytimes.android.eventtracker.KEY_SESSION_TIME_VALUE"
    value="1590555752362" />
    <int name="LONGEST_STREAK" value="0" />
    <boolean name="show_ab_group_one_key" value="false" />
    <boolean name="FIRST_LAUNCH" value="true" />
    <boolean name="SKIP_FILLED_SQUARES" value="true" />
    <boolean name="PLAY_VICTORY_JINGLE" value="true" />
</map>
marlin:/data/user/0/com.nytimes.crossword/shared_prefs #
```

You will see how Android designates the type of each of the key-value pairs as well. You can see strings, Booleans, long, and int values in the XML file. By altering these key-value pairs, you can, in most cases, affect the way the app works.

The databases and `shared_prefs` directories are usually places that I go to frequently because they contain a wealth of data that not only tells you about the inner workings of the app but also gives you an opportunity to change how an app behaves.

Detecting and Hiding Root

Rooting your device is all well and good, but you must keep in mind that there are others as well who will be rooting their devices. Then, when your apps run on those devices, they remain at the mercy of whatever the rooted device's owner throws at it. Therefore, it makes sense to learn about detecting if a device is rooted or not. Banking applications often rely on root detection of Android devices. If a rooted device is detected, then the app will quit before any further communication or interaction takes place.

The mechanisms of detecting root are not exact. It comes down, once again, to a cat and mouse game where the app developer has to account for and be aware of all the signatures of a rooted device. These can be broadly divided into the following categories:

- Looking for specific files that have been installed, for example, looking for the su command or even attempting to execute it. Looking for directories that may not originally exist on the Android system partition. The SuperSU root package would create an `xbin` directory.
- Looking for installed packages by listing all the packages and looking for known names like `com.topjohnwu.magisk` for Magisk or other known root packages. Another thing that is done is a substring comparison where the detector will search for certain words in packages or in the filesystem.
- Looking for test keys that Google ships with their testing or developer units. This is normally done by checking build tags on Android.
- Looking for a writeable /system partition. The `/system` partition is not meant to be writeable and is always mounted read-only on non-rooted devices.

In this section, I won't go into too much depth of the many exhaustive ways of checking for root. I will, however, take a closer look at a package called RootBeer [https://github.com/scottyab/rootbeer] which you can add to your app and call its root detection functions. RootBeer was developed by Scott Alexander-Brown and Mat Rollings. Let's go ahead and add that to our aas2obfuscate app. Add this line to the app's `build.gradle` file:

```
implementation 'com.scottyab:rootbeer-lib:0.0.8'
```

Mine looks like this:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_
    version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.scottyab:rootbeer-lib:0.0.8'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Then, edit your `MainActivity.kt` and add this bit of code to the end of the file and run it:

```
val rootBeer = RootBeer(applicationContext)
if (rootBeer.isRooted()) {
    Log.d("aas2obfuscate","Device has been rooted!")
} else {
    Log.d("aas2obfuscate", "No root detected")
}
```

You should be able to see in your logcat that RootBeer instantly detects that Magisk is installed, and then our logging line is called:

```
E/RootBeer: RootBeer: isAnyPackageFromListInstalled() [249] - com.
topjohnwu.magisk ROOT management app detected!
E/QLog: RootBeer: isAnyPackageFromListInstalled() [249] - com.topjohnwu.
magisk ROOT management app detected!
D/aas2obfuscate: Device has been rooted!
```

Defeating Root Detection

As is the norm with apps that root Android devices, they typically ship with a root cloaking or hiding app. Magisk is no exception and ships with a program called `magiskhide`. Magiskhide is disabled by default, and you have to enable it through the command line. Let's examine its options first:

```
marlin:/system # magiskhide
MagiskHide 20.4(20400)

Usage: magiskhide [action [arguments...] ]

Actions:
   status          Return the status of magiskhide
   enable          Start magiskhide
   disable         Stop magiskhide
   add PKG [PROC]  Add a new target to the hide list
   rm PKG [PROC]   Remove target(s) from the hide list
   ls              Print the current hide list
   exec CMDs...    Execute commands in isolated mount
                   namespace and do all hide unmounts
1|marlin:/system #
```

To check if magiskhide is enabled, you can do a `magiskhide status`. Similarly, execute `magiskhide enable` to enable it. Magiskhide also maintains a list of packages that you can add to. This is known as the hide list and will take some extra measures to hide apps that specifically do root detection. Let's now hide our device's rooted status. If we look at the logs, we can see that RootBeer is detecting the Magisk Manager. This is likely by looking at the installed packages. So let's tackle that first by hiding the Magisk Manager. On your device, open Magisk Manager and go into settings. You will then see an option called Hide Magisk Manager as shown in Figure 8-20. Click that and it will prompt you to give Magisk Manager an alternate name (Figure 8-21). I chose Policy. After I click the OK button, Magisk will repackage itself and name itself by the name that you chose.

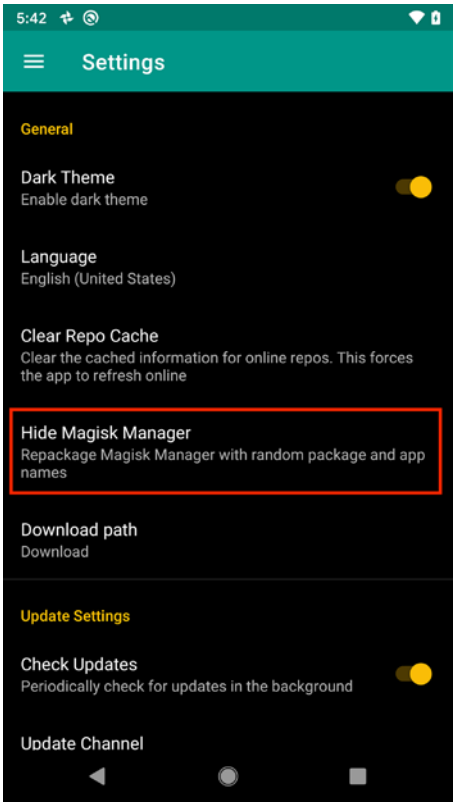


Figure 8-20. *The setting to hide Magisk Manager*

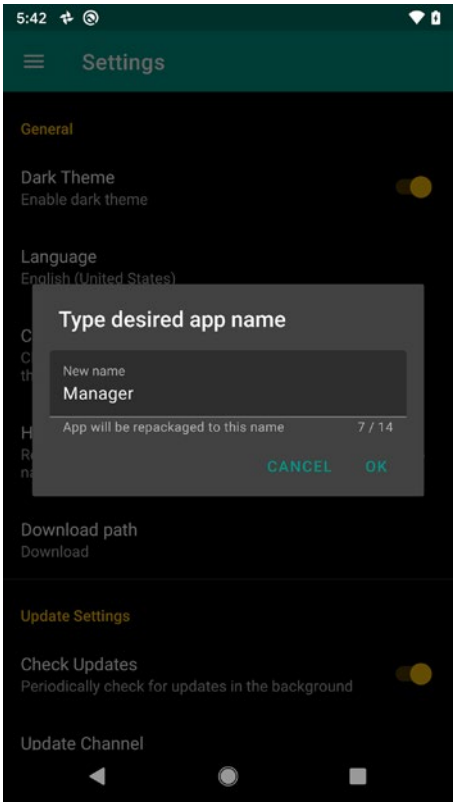


Figure 8-21. *Renaming the Magisk Manager from Manager to a name of your choosing*

Now let's see if our app still continues to detect Magisk Manager. Run it again and look at the logs:

```
V/RootBeer: RootBeer: checkForBinary() [194] - /sbin/su binary detected!
W/2.aas2obfuscate: type=1400 audit(0.0:3058): avc: denied { read } for
name="cache" dev="sda34" ino=16 scontext=u:r:untrusted_app:s0:c187,c256,
c512,c768 tcontext=u:object_r:cache_file:s0 tclass=lnk_file permissive=0
D/aas2obfuscate: Device has been rooted!
```

Hmm, looks like it's a different message now. Right now, RootBeer doesn't detect the Magisk Manager which is great, but it does detect that we have the /sbin/su binary file on our filesystem. Let's see if adding our app to the magiskhide list takes care of it. As we know, we named our app `com.redteamlife.aas2.aas2obfuscate`. So, let's go ahead and add that to `magiskhide`:

```
1|marlin:/data/local/tmp # magiskhide enable
marlin:/data/local/tmp # magiskhide ls
com.google.android.gms|com.google.android.gms.unstable
org.microg.gms.droidguard|com.google.android.gms.unstable
marlin:/data/local/tmp # magiskhide add com.redteamlife.aas2.aas2obfuscate
marlin:/data/local/tmp # magiskhide ls
com.google.android.gms|com.google.android.gms.unstable
com.redteamlife.aas2.aas2obfuscate|com.redteamlife.aas2.aas2obfuscate
org.microg.gms.droidguard|com.google.android.gms.unstable
marlin:/data/local/tmp #
```

Here, I first enable `magiskhide` and then add our app's bundle id to the hide list. I do a quick `magiskhide ls` to list the packages on the list to verify that our package has been added. Now, let's test one more time:

```
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/data/local/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/data/local/bin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/data/local/xbin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/sbin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/su/bin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/system/bin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/system/bin/.ext/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/system/bin/failsafe/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/system/sd/xbin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/system/usr/we-need-root/su Absent :(
```

```
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/system/xbin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/cache/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/data/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/dev/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/system/sbin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/product/bin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/apex/com.android.runtime/bin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/odm/bin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/vendor/bin/su Absent :(
2020-05-27 17:56:10.038 11218-11218/? I/RootBeer: LOOKING FOR BINARY:
/vendor/xbin/su Absent :(
2020-05-27 17:56:10.041 11218-11218/? D/aas2obfuscate: No root detected
```

Boom! No root detected! Our `magiskhide` cloaking did the trick. So, what have we learned? Well, once again, as the root user, you are all powerful. You can craft the entire look and feel of the device masking processes that you don't want visible in however way you want to. The apps that run on the device that is rooted usually don't stand a chance – yet another example of why you want to rethink what data you store on the device through your app.

Next, I want to write a script using Frida to see if we can fool RootBeer into thinking we are not rooted. To do that, we should first check the source code for RootBeer. We already know that all we have to do is call the `isRooted()` method. Let's see what that method looks like:

```
public boolean isRooted() {

        return detectRootManagementApps() ||
        detectPotentiallyDangerousApps() || checkForBinary(BINARY_SU)
                || checkForDangerousProps() || checkForRWPaths()
                || detectTestKeys() || checkSuExists() ||
                checkForRootNative() || checkForMagiskBinary();
    }
```

This function will call all other functions which return either a true or false. Any one of them returning false will make the entire method return a false. So for us to intercept and neutralize this, we have to reimplement the `isRooted()` method. Here is my JavaScript code to do that:

```
1: setTimeout(function() {
2:     Java.perform(function () {
3:          const RootBeer = Java.use('com.scottyab.rootbeer.RootBeer');
4:          RootBeer.isRooted.implementation = function(){
5:               return false;
6:          }
7:     });
8: });
```

You can see it is very simple to reimplement the `isRooted()` method to always return false. Save this file in your Frida directory as rootbeer.js. First reverse the magiskhide changes that you made above so that it does not interfere with the process using Frida. Now making sure Frida server is running on your device, launch the aas2obfuscate app through Frida by executing this command on your workstation:

```
(p3) ➜  frida -U -f com.redteamlife.aas2.aas2obfuscate -l rootbeer.js
        --no-pause

     ____
    / _  |   Frida 12.9.4 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
```

```
   . . . .   More info at https://www.frida.re/docs/home/
Spawned `com.redteamlife.aas2.aas2obfuscate`. Resuming main thread!
[Pixel XL::com.redteamlife.aas2.aas2obfuscate]->
```

The aas2obfuscate app should start. Now run logcat and filter for the tag aas2obfuscate so that you can see what is written out to the log. On your device shell, run `logcat aas2obfuscate`:

```
05-27 18:13:13.516 11862 11862 W 2.aas2obfuscat: Accessing hidden
method Landroid/view/View;->computeFitSystemWindows(Landroid/graphics/
Rect;Landroid/graphics/Rect;)Z (greylist, reflection, allowed)
05-27 18:13:13.516 11862 11862 W 2.aas2obfuscat: Accessing hidden method
Landroid/view/ViewGroup;->makeOptionalFitsSystemWindows()V (greylist,
reflection, allowed)
05-27 18:13:13.584 11862 11862 D aas2obfuscate: BT Adapter address is
02:00:00:00:00:00
05-27 18:13:13.584 11862 11862 D aas2obfuscate: No root detected
05-27 18:13:13.642 11862 11907 I Adreno  : QUALCOMM build
: 4a00b69, I4e7e888065
05-27 18:13:13.642 11862 11907 I Adreno  : Build Date
: 04/09/19
05-27 18:13:13.642 11862 11907 I Adreno  : OpenGL ES Shader Compiler
Version: EV031.26.06.00
```

There you have it. The line in bold shows that RootBeer has not detected that the device has been rooted.

Further Tools to Help Debugging

I want to wrap this chapter up by talking about a few more tools that you can use when you want to see the inner workings of apps. One of them is `strace` and the other is `jtrace`. Now `strace` is a popular Linux debugging program that allows you to see all the syscalls that a program makes. For example, remember that Crossword app we downloaded and installed? If we wanted to see which files it opened during its execution, we could see that with `strace`. Here's the bad news though. `strace` does

not come bundled with my Google Pixel XL, and I think it may also be left out of other vendor devices. All is not lot though, because we can build it and copy it across to use as we like. First, let's build `strace`. We are going to use Docker for this, and since you already installed and used it in a previous chapter, you should not be too deep in the water with this. Remember, since we're building `strace` for the ARM64 platform, we will have to cross-compile it. We can use the Ubuntu Docker image for our example. I am working in a separate directory that I use for third-party code. I will be mounting this directory and mapping it to my Docker container later. Let's start by pulling the correct image:

```
➜  docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
Digest: sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5
        c54d7
Status: Image is up to date for ubuntu:latest
docker.io/library/ubuntu:latest
➜
```

With the image downloaded, let's next go and get the `strace` source code. `strace` has a home at https://strace.io/. You can find and download the source code from here: https://strace.io/files/5.6/strace-5.6.tar.xz. Download it into a directory of your choosing and `untar` it as shown. Here, I'm doing it in my code directory:

```
➜  tar xf strace-5.6.tar.xz
➜  cd strace-5.6
➜
```

Now let's fire up our Docker container and mount the `strace` directory to a directory on the container. Run this command:

```
docker run --rm -it -v "<replace with your directory name>":"/strace"
ubuntu
root@468d83dd9d34:/#
```

Check that the strace directory was mounted correctly:

```
root@468d83dd9d34:/# ls /strace
AUTHORS                                          ioprio.c
ptp.c
COPYING                                          ipc.c
ptrace.h
CREDITS.in                                       ipc_defs.h
ptrace_syscall_info.c
...
...
```

Now we will need to update our Ubuntu image and install the tools necessary to build programs on it. To do this, run the following two commands:

```
root@468d83dd9d34:/#  apt-get update
Get:1 http://ports.ubuntu.com/ubuntu-ports focal InRelease [265 kB]
Get:2 http://ports.ubuntu.com/ubuntu-ports focal-updates InRelease [107 kB]
Get:3 http://ports.ubuntu.com/ubuntu-ports focal-backports InRelease [98.3 kB]
. . .
. . .
. . .
```

and

```
root@468d83dd9d34:/# apt-get install -y build-essential gcc-aarch64-linux-
gnu binutils-aarch64-linux-gnu
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  binutils binutils-aarch64-linux-gnu binutils-common cpp cpp-9 dirmngr
  dpkg-dev fakeroot g++ g++-9 gcc gcc-9
  gcc-9-base gnupg gnupg-l10n gnupg-utils gpg . . .
. . .
. . .
. . .
```

After this completes, it's time to build `strace`. To do this, we will first have to configure the environment using the configure command. Change directory to the `strace` directory and then run this command:

```
root@468d83dd9d34:/strace# ./configure --enable-mpers=no --host aarch64-
linux-gnu LDFLAGS="-static -pthread"
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /usr/bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
. . .
. . .
. . .
```

This will build the relevant configuration files necessary for the next step which is to build it. To do this, we simply run make:

```
root@468d83dd9d34:/strace# make
```

This will build the `strace` binary in the current directory. You can now copy this across to your device from your workstation:

```
➜  adb push strace /data/local/tmp
```

We're now ready to see our shiny new `strace` in action. We were planning to see what the Crossword app did as far as files were concerned. To do this, let's trace the "`openat`" syscall (typically you may want to start with the "`open`" syscall, but from prior knowledge of this app, I know it uses "`openat`"). Start the Crossword app, but then close it by pressing the back button until you exit the app. This will leave it running in the background. Let's get its process id first. On your Android shell, run

```
marlin:/data/local/tmp # ps -ef|grep crossword
root         16026 12548 2 20:43:23 pts/0 00:00:00 grep crossword
u0_a180      16374   649 0 13:58:50 ?     00:00:26 com.nytimes.crossword
marlin:/data/local/tmp #
```

In my case, the process id is 16374. From your /data/local/tmp directory, start strace with the following parameters:

```
marlin:/data/local/tmp # ./strace -e trace=openat -f -p 16374
./strace: Process 16374 attached with 89 threads
pid 16859] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/databases/
google_app_measurement_local.db", O_RDWR|O_CREAT|O_LARGEFILE|O_CLOEXEC,
0600) = 116
[pid 16430] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/
databases", O_RDONLY|O_CLOEXEC) = 119
[pid 16592] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/no_backup/
.localytics/com.localytics.android.6f88eb5437a02c5a32a000dcd15ab6f01e46668255
c960679c11961f455e5a5e.analytics.sqlite-journal", O_RDWR|O_CREAT|O_LARGEFILE|
O_CLOEXEC, 0660) = 119
[pid 16859] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/databases/
google_app_measurement_local.db-journal", O_RDWR|O_CREAT|O_LARGEFILE|
O_CLOEXEC, 0660) = 117
[pid 16859] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/
databases", O_RDONLY|O_CLOEXEC) = 236
[pid 16398] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/
shared_prefs/com.nytimes.crossword_preferences.xml", O_WRONLY|O_CREAT|
O_TRUNC, 0600) = 116
[pid 16398] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/shared_prefs/
androidx.work.util.id.xml", O_WRONLY|O_CREAT|O_TRUNC, 0600) = 116
[pid 16398] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/shared_prefs/
EntitlementsAndPurchase.xml", O_WRONLY|O_CREAT|O_TRUNC, 0600) = 116
[pid 16398] openat(AT_FDCWD, "/data/user/0/com.nytimes.crossword/shared_prefs/
com.google.android.gms.measurement.prefs.xml", O_WRONLY|O_CREAT|O_TRUNC, 0600
<unfinished ...>
. . .
. . .
```

Breaking down the parameters we used for strace, the -e trace=openat means we want to only trace the openat syscall, the -f means follow any child processes or threads that the original app creates, and the -p is the PID or process id of the app. Using this

approach, you can examine what apps installed on the device are doing, where they read from or write to, and which network hosts they communicate with.

One more app that's similar to `strace` but is more Android aware is `jtrace`. `jtrace` [http://newandroidbook.com/tools/jtrace.html] is an app that is written by Jonathan Levin. Jonathan Levin is well known for his research in iPhone and Android internals, and his books are an absolute must read for anyone interested in the inner workings of Android. He developed `jtrace` to be more Android aware. It works in much the same way as `strace`. To get it running on your device, just download the archive, decompress it, and copy over the 64 bit ARM version of `jtrace` to your `/data/local/tmp` directory and run it from there. It takes the same `-f` and `-p` flags.

Summary

This chapter was a bit of a heavy one, but I feel like I have only scratched the surface of the rooting topic. For the purposes of this book, I will not go into further depth, but I will look toward covering a bit more about rooting in this book's companion my site [https://aas2book.com].

The rooting process is an essential technique to learn if you are to truly move to testing your applications in depth. With a rooted device, you can really explore the full reaches of your device and then subject your app to stresses and attacks that can help to further strengthen it. With device rooting being made almost point and click, and the tools for breaking SSL Pinning or breaking encryption are again very much within reach of many, it makes sense to go this far in your security testing.

CHAPTER 9

Bypassing SSL Pinning

With so much discussion around the topic of securing data in transit, I wanted to take a bit of time to dive into the actual process of SSL/TLS encryption and how Android and apps written for Android handle this. The best way to go about it is to get down and do the work, so in this chapter, we will see how to generate an SSL certificate, write a back-end API in Golang, and write an Android client to talk to that back end, and finally we will see how to intercept SSL traffic.

Let's first take a very quick look at how an SSL connection is established. Figure 9-1 shows the steps that take place after TCP connectivity has been established. Both the client and server go through a handshake process where they exchange certificates and what encryption ciphers they both use. They agree on a cipher and then proceed.

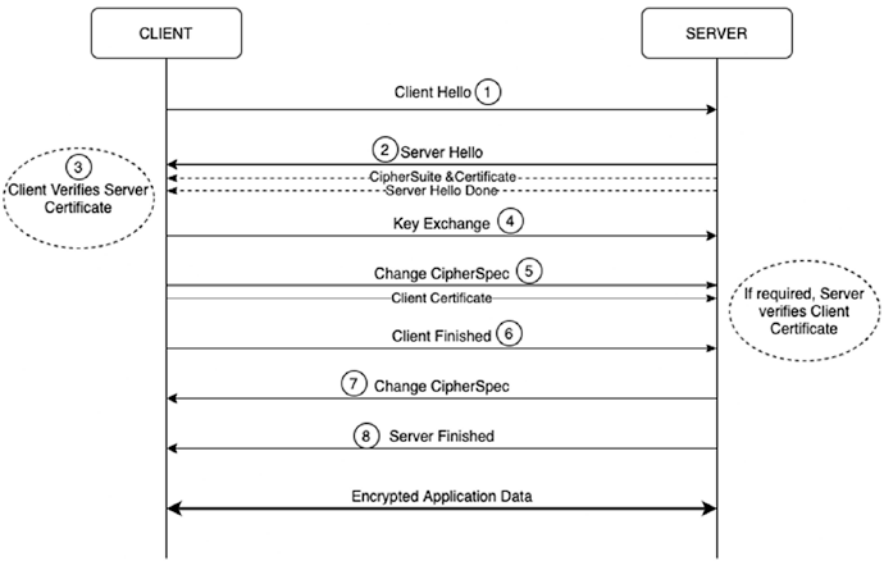


Figure 9-1. *What happens during an SSL handshake*

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_9

Take note of step 3 in the figure. This is where the client will verify the certificate chain of the server certificate presented. Further, it may also choose to verify the hostname of the server. For example, if the server certificate shows `www.example.com`, but you access the server using `https://192.168.10.10`, then because there's a hostname mismatch, the client can alert the user that something is not right. To see this in action, if you're keen to, you can always try using OpenSSL on your command line. Pick your favorite SSL website (I am using my own site `aas2.redteamlife.com` for this example) and then type in the following on your terminal command line:

```
openssl s_client -connect aas2.redteamlife.com:8443
CONNECTED(00000005)
depth=2 O = Digital Signature Trust Co., CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
verify return:1
depth=0 CN = aas2.redteamlife.com
verify return:1
---
Certificate chain
 0 s:/CN=aas2.redteamlife.com
   i:/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
 1 s:/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
   i:/O=Digital Signature Trust Co./CN=DST Root CA X3
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIFYTCCBEmgAwIBAgISA55ixGZHoNROz2Mrr9bnzH5dMA0GCSqGSIb3DQEBCwUA
. . .
. . .
. . . Trimmed
. . .
. . .

TBrIULhzOqXGOq67DPityZYgLwtyCusImsiZNqsdRPfwcY/NiCOZWrf+15I2yfsD
/lKHRZNHY+1GNHRe/Zklf1ZOt3vsY2Md4OnvDXkknJDwMfpzzg==
-----END CERTIFICATE-----
subject=/CN=aas2.redteamlife.com
```

```
issuer=/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
---
No client certificate CA names sent
Server Temp Key: ECDH, X25519, 253 bits
---
SSL handshake has read 3137 bytes and written 289 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID: D938B83987BD43E6C7F62FDBFFBAE0E0C8C5269B543D6F572AFDD689BB2D9A73
    Session-ID-ctx:
    Master-Key: B945C0A53797EE5D2B16106115BC777A0805B9F1EA13EA457110C8E9CFC
                0F12BAC9052AA30C16D057D8E7064EB3035FD
    TLS session ticket:
    0000 - a3 25 0e a8 18 4a 49 13-92 4c 5f 9b ca 32 fc 6c   .%...JI..L_..2.l
    0010 - 20 49 3f 9f e4 b6 8a 37-0f c9 88 3a 07 2d 21 9d    I?....7...:.-!.
    0020 - fd d8 12 37 4b 11 46 9c-1c 9b e0 dd 82 81 9f 9d   ...7K.F.........
    0030 - 2b e3 40 8c 7a 88 fe a1-ed 7a 67 52 e6 fb 03 e6   +.@.z....zgR....
    0040 - ff 27 98 bc 68 02 ac 7c-53 b0 af 29 1e 3e 6e e5   .'..h..|S..).>n.
    0050 - 5b 10 e7 8d 18 da f4 6a-0b b5 b1 fb f6 76 5d 1b   [......j.....v].
    0060 - b8 46 ce 39 6a 05 e5 08-81 0e 39 24 13 96 87 d4   .F.9j.....9$....
    0070 - 01 64 89 cb 48 46 ea 27-                          .d..HF.'

    Start Time: 1589256227
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
DONE
```

If you're used to using telnet or nc to verify HTTP, the way to do that with an SSL-enabled site is to use the OpenSSL command `s_client`. In the preceding code block, after the DONE prompt, you can type in normal HTTP requests like `GET / HTTP/1.1` and so on.

SSL Certificates

Generally, an SSL certificate is issued by a Certificate Authority or CA. A CA's job, among issuing you a certificate, is to verify your identity. The CA is what vouches for you when other clients ask about how trustworthy you are. Once a certificate is issued by a CA, this is one mechanism in which they say "Yeah this guy's shown us all his docs and we believe he's legit so trust us when you see that his certificate is issued by us." In the early days of public key infrastructure (PKI), obtaining an SSL certificate was quite a process. The process is a lot more straightforward now as you will see. But first, let's talk about the three types of validation for HTTP certificates.

Domain Validation

Domain Validation (DV) certificates do one thing. They verify that the person applying for the certificate is the person who also controls and has ownership over the name on the certificate that he is applying for. We apply for one later on in this section. A DV certificate can be the quickest to register for and download because it is very easy to validate ownership of a server as you will see. Therefore, no further human interaction is involved on the CA's side.

Organizational Validation

An Organizational Validation (OV) certificate, on the other hand, will check not just for ownership of the server as a DV certificate but will also request for business entity registration documents to prove that the person that is purchasing the certificate can manage the domain name as well as proving that the company is real. This one would require some human inspection and additional checking on the CA's side. Of course these procedures depend from CA to CA, but it is safe to assume that the bigger more trusted CAs will do the relevant due diligence as far as checking business registration before issuing the certificate.

Extended Validation

The Extended Validation (EV) certificate, you may have heard the term EVSSL being mentioned by people, is the certificate you want to go to if you're doing an online business that requires you to verify to your customers that you jumped through the relevant hoops to get this certificate. An EV certificate will typically turn modern browser address bars green and show a heightened level of trust visually. To get an EV certificate, the CA will always want to speak with a human from the company. The checks in place are the ones for an OV certificate, but also include a mandatory human-to-human session where the CA staff can verify further details of your business and its legitimacy prior to issuing you a certificate. Generally, you will find that ecommerce companies or companies that may do online financial transactions will get EV certificates.

Self-Signed Certificates

Another quick way in which you can generate a server certificate is to issue one yourself. These are called self-signed certificates and can be generated by using OpenSSL. The command to do this and accompanying output are as follows:

```
> openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365
Generating a 4096 bit RSA private key
..............................++
...++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:SG
State or Province Name (full name) []:Singapore
```

```
Locality Name (eg, city) []:Jurong
Organization Name (eg, company) []:Madison Technologies
Organizational Unit Name (eg, section) []:
Common Name (eg, fully qualified host name) []:www.redteamlife.com
Email Address []:sheran@example.com
> ls
cert.pem key.pem
>
```

The command `openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365` will generate a new certificate valid for 365 days. Pay close attention to the Common Name. This is the name of your domain which you want the certificate issued for. At the end of this process, you will see a certificate and private key which you can then take and use in your web servers and so on. However, note that if you try to access a web server that has a self-signed certificate, your browser will likely show you an error or warning screen such as the one in Figure 9-2. If you click the "Not Secure" portion, your browser will show you more information about why the certificate was not trusted in Figure 9-3.

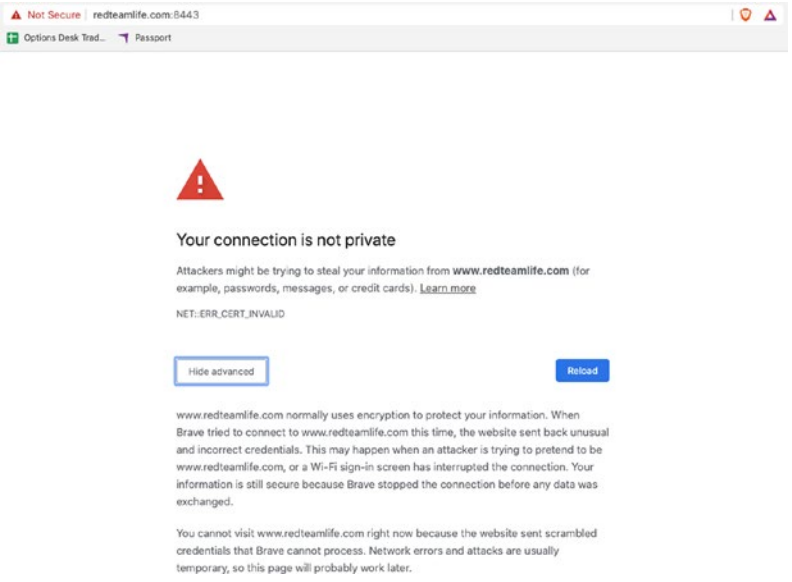


Figure 9-2. *The error screen when browsing to a site with an untrusted certificate*

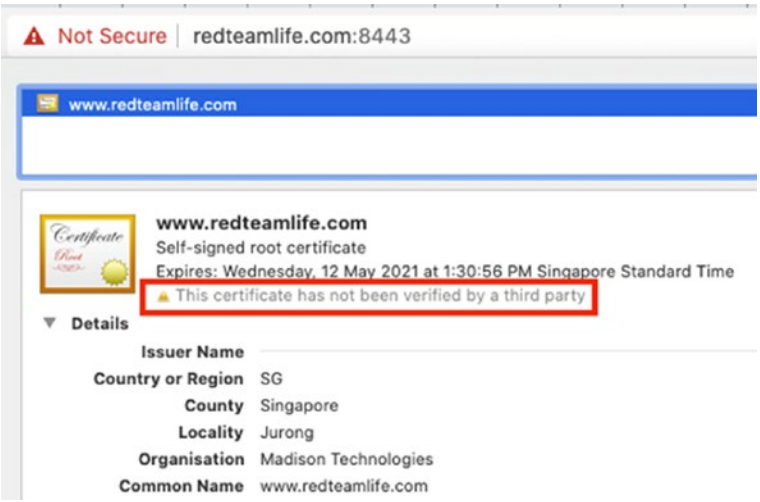


Figure 9-3. *The reason why the certificate was not trusted*

A Note About Verification

Why does the preceding example fail? I mean we have a certificate and private key just like we should. Well, the reason this fails or at least the browser doesn't honor the certificate is because it doesn't trust the person issuing it. Essentially, it is like you drawing your own ID card and trying to use it when you buy alcohol. The guy selling it to you, in most cases, will laugh in your face because he doesn't trust the ID card. He would trust the government that issues the ID card, and he already knows what a government-issued ID card looks like. Your ID card looks nothing like the government one, and hence you wouldn't be able to get that bottle of Single Malt.

You may then wonder how a client can begin to trust a CA's issued certificate. Well, each of our web clients these days and even operating systems come with a set of root certificates. A root certificate is a self-signed certificate that a CA issues to itself. Based on this certificate, it will sign all other certificates that it issues. But wait, that's the same process we used right? Well yes, that's exactly what we used to generate our root certificate. The difference here is that the bigger CAs gain a reputation first and then begin to gain trust. CAs may ask all clients to include their root certificate in the distribution of the latest browser or operating system. Or a client may choose to bundle the root certificate of the CAs that it believes are more trustworthy. Either way, the entity that controls whether a certain root CA is included or not is the publisher of the browser or operating system. We have grown to know and trust the bigger names that make browsers like Firefox, Safari, or Chrome so we use them. This in itself becomes

an attack vector. For example, if I wanted to sniff all your SSL web traffic easily, I would embed my own root certificate in a browser that I build (technically, if I am writing the browser, then I don't even have to implement SSL, but let's assume for argument's sake that I base my browser off Chrome). Then whenever I wish to sniff traffic, I can poison or redirect a legitimate SSL host to a server that belongs to me, issue a certificate for that legitimate site, and begin to collect SSL encrypted traffic. The browser won't report an error because it sees that the certificate for that site was generated and signed by my root certificate. Since the root certificate is in the browser's repository of certificates, it must be legitimate. The topic of SSL and encryption can take up a book by itself, so I am going to quickly move on to our main topic of how to break SSL.

Getting a DV Certificate

To get ourselves a DV certificate, we're going to use Let's Encrypt [https://letsencrypt.org/]. Let's Encrypt is a nonprofit CA that provides DV certificates for free. The service itself is provided by the Internet Security Research Group that had board members and technical advisors from many of the large companies such as Google, Facebook, and Akamai and from other groups such as the ACLU and EFF. The entire process is automated as we will see shortly. For the issuing of a DV certificate, remember that we have to prove that we own the domain name and the server it is running on? This also means that we need to have a domain name and a server to run it on that can be publicly accessible. If you want to follow along, that means you will need a public IP that is configured on a server or workstation of yours that you have access to. I used DigitalOcean [https://www.digitalocean.com/] and quickly spun up a Debian Linux virtual machine or droplet to run this test. You could also choose to use Ubuntu which is based off of Debian. For the demo, I am going to use my domain name called redteamlife.com and will apply for a subdomain name certificate: aas2.redteamlife.com. OK, let's get started.

You will first need to have a web server running. So let's first get started with that. We will use nginx for this demo. First, let's update our version of Debian so we can download the latest packages. On your root shell, run

```
# apt-get update
```

Then once you have updated all your packages, install nginx like so:

```
# apt install nginx
```

Answer yes to when it asks if you want to continue and wait for nginx to be installed. At this point, I am going to assume your domain name is pointing to the IP address of the server that is running the nginx server. Test your nginx server to see if it is reachable by entering your domain name in your browser. You should see something that resembles Figure 9-4.

Figure 9-4. *The nginx default page when accessed from a browser*

Now we need to use the Certbot to generate our certificate.

Certbot

Certbot [`https://certbot.eff.org/`] is the EFF's automated certificate generator. The best way to get a certificate quickly and painlessly, in my opinion, is to use a popular Linux distro with shell access. The reason I say use a popular distro is because the Certbot instructions are a lot easier to follow and generally more precise.

Having stated this, I am aware that getting Certbot up and running well can be a little bit tricky. Here is a list of key points that you need for getting Certbot up and running fast:

1. A server with a public IP address
2. A domain name purchased and correctly pointing to the IP address from point 1
3. Ports 80 and 443 open on the server from point 1
4. Ability to install and configure a web server properly (e.g., nginx) and pick a server operating system where you can install nginx smoothly

5. There are specific instructions that can be found in the Certbot website that gives you a very good combination of web servers and operating systems to choose from. For further information, you can visit this URL: https://certbot.eff.org/instructions.

On the Certbot page, select which web server and operating system that you use. Figure 9-5 shows you what that looks like. I have selected nginx and Debian 10 to match my setup.

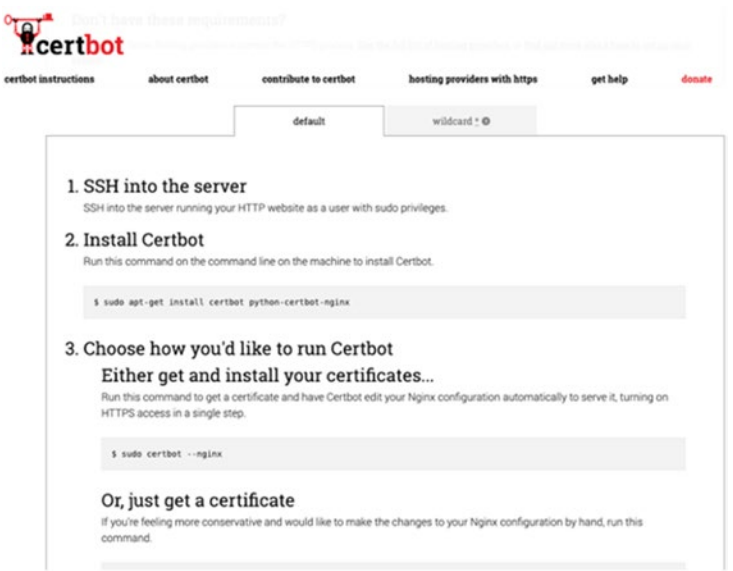


Figure 9-5. *Certbot gives you the instructions you need to get setup with your certificate*

I'm going to save you from having to look at any more screenshots for this, but I followed the instructions as stated and ended up with my certificate and private key file like from the preceding openssl example. Let's Encrypt will usually write your certificates to `/etc/letsencrypt/live.<domain name>/`.

The Back End

We could technically conduct our demo by using the nginx web server that we just installed and requested a certificate for, but I want to show you a bit more on how API communications will look on the wire, so let's go ahead and write a back end. I will give you the basic blueprint of the back-end server so you can implement it in the language of your choice. I chose Golang (https://golang.org/) and we will look at the source code later on. You can download the source code here: https://github.com/sheran/aas2-ch09-go_backend.

Note that you will need both the certificate and private key file in the directory that you run this file from.

Back-End Server Specification

1. The server has to listen on port 8443.
2. The server has to implement and use SSL from the certificates we generated.
3. The server should have one API called /secret that accepts HTTP POST requests only.
4. The /secret API should receive JSON data with specification {"code":<string>} where string is sent from the client.
5. The server should return {"message":<message of your choice if code is incorrect>} if an incorrect code is set.
6. The server should return {"message":<message of your choice if code is correct} if the correct code is sent.
7. The code can be a string of your choosing.

Here is the Golang code that implements the preceding specification:

```
01: package main
02:
03: import (
04:       "encoding/json"
05:       "fmt"
```

```
06:       "log"
07:       "net/http"
08: )
09:
10: type ReqStruct struct{
11:       Code string `json:"code"`
12: }
13:
14: type ResStruct struct{
15:       Message string `json:"message"`
16: }
17:
18:
19: func greeter(w http.ResponseWriter, req *http.Request){
20:       fmt.Fprintf(w,"Hello!")
21: }
22:
23: func secret(w http.ResponseWriter, req *http.Request){
24:       if req.Method == http.MethodPost{
25:             var jsonData ReqStruct
26:             if err := json.NewDecoder(req.Body).Decode(&jsonData); err != nil{
27:                   http.Error(w,err.Error(),http.StatusInternalServerError)
28:                   return
29:             }
30:             if jsonData.Code == "gekko"{
31:                   response := &ResStruct{Message: "Blue Horseshoe Loves Anacott
                      Steel"}
32:                   if err := json.NewEncoder(w).Encode(response); err != nil{
33:                       http.Error(w,err.Error(),http.StatusInternalServerError)
34:                       return
35:                   }
36:             } else {
37:                   response := &ResStruct{Message: "Wrong code, the SEC is on
                      the way to you now."}
```

```
38:                 if err := json.NewEncoder(w).Encode(response); err != nil{
39:                     http.Error(w,err.Error(),http.StatusInternalServerError)
40:                     return
41:                 }
42:             }
43:     }
44: }
45:
46: func main(){
47:     http.HandleFunc("/greeting",greeter)
48:     http.HandleFunc("/secret",secret)
49: log.Fatal(http.ListenAndServeTLS(":8443","fullchain.pem","privkey.pem",nil))
50: }
```

Let's go over the specification.

Line 49 addresses both Spec 1 and Spec 2. My certificate and private key are in my current directory named fullchain.pem and privkey.pem, respectively.

Lines 23–44 and 48 address Spec 3 and Spec 4.

Lines 37–41 address Spec 5.

Lines 30–35 address Spec 6.

And I've chosen the code of "gekko".

After I deploy and run my back end, I can test it by using curl:

```
➜ curl -X POST -d '{"code":"arglebargle"}' https://aas2.redteamlife.
  com:8443/secret
{"message":"Wrong code, the SEC is on the way to you now."}

➜ curl -X POST -d '{"code":"gekko"}' https://aas2.redteamlife.com:8443/secret
{"message":"Blue Horseshoe Loves Anacott Steel"}

➜
```

In this example, I use the hostname aas2.redteamlife.com. You will have to use the domain name that you selected for this exercise.

It seems to be working just fine, which is great. Next, let's build an Android client to talk to that API.

Android Client

Similar to how we did in Chapter 3, we're going to build a very basic Android client that has just one purpose – to capture our input and send it to the back-end api, receive a response, and print it out on the screen. In Android Studio, start a new Android Studio Project and then pick the one with an Empty Activity. Name it accordingly as you would want to (refer to Chapter 3) and let's design a UI.

Here's my design in Figure 9-6.

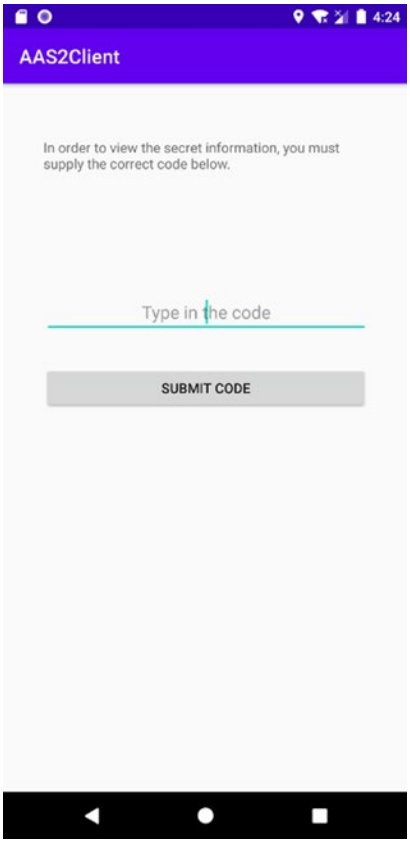


Figure 9-6. *The Activity design of the client app*

I am just going to give you my strings.xml, activity_main.xml, MainActivity.kt, and NetUtils.kt source here. These are the files necessary to build the app. You can find the full source code to this client here: https://github.com/sheran/aas2-ch09-android_client_1.

The strings.xml file

```
<resources>
    <string name="app_name">AAS2Client</string>
    <string name="loader">[Code Appears Here]</string>
    <string name="instructions">In order to view the secret information,
    you must supply the correct code below.</string>
    <string name="submit">Submit Code</string>
    <string name="codePH">Type in the code</string>
</resources>
```

The activity_main.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/loaderText"
        android:layout_width="330dp"
        android:layout_height="45dp"
        android:textAlignment="center"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.217" />
```

```
    <TextView
        android:id="@+id/textView"
        android:layout_width="330dp"
        android:layout_height="85dp"
        android:clickable="false"
        android:text="@string/instructions"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.087" />

    <Button
        android:id="@+id/button"
        android:layout_width="330dp"
        android:layout_height="45dp"
        android:text="@string/submit"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.425" />

    <EditText
        android:id="@+id/editText"
        android:layout_width="330dp"
        android:layout_height="45dp"
        android:ems="10"
        android:hint="@string/codePH"
        android:inputType="text"
        android:textAlignment="center"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.506"
        app:layout_constraintStart_toStartOf="parent"
```

```
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.311" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

The MainActivity.kt file

```
01: package com.redteamlife.aas2.aas2client
02:
03:
04: import androidx.appcompat.app.AppCompatActivity
05: import android.os.Bundle
06: import android.widget.Button
07: import android.widget.EditText
08: import android.widget.TextView
09:
10: class MainActivity : AppCompatActivity() {
11:     override fun onCreate(savedInstanceState: Bundle?) {
12:         super.onCreate(savedInstanceState)
13:         setContentView(R.layout.activity_main)
14:         val button : Button = findViewById(R.id.button)
15:         button.setOnClickListener{
16:             val text : EditText = findViewById(R.id.editText)
17:             if(!text.text.isEmpty()){
18:                 val networkTask = NetworkAsyncTask(this)
19:                 networkTask.execute("https://aas2.redteamlife.com:8443/
                    secret",text.text.toString())
20:             }
21:         }
22:     }
23:
24:     fun setText(text: String){
25:         var loader: TextView = findViewById(R.id.loaderText)
26:         loader.text = text
27:     }
28:
```

```
29: }
30:
```

The NetUtils.kt file placed in the same directory as your MainActivity.kt file

```
01: package com.redteamlife.aas2.aas2client
02:
03: import android.os.AsyncTask
04:
05: import org.json.JSONObject
06: import java.io.*
07: import java.net.URL
08: import javax.net.ssl.HttpsURLConnection
09:
10: public class NetworkAsyncTask(activity: MainActivity):
    AsyncTask<String, Void, String>(){
11:
12:     private val mActivity = activity
13:
14:     override fun doInBackground(vararg params: String?): String? {
15:         var connection: HttpsURLConnection? = null
16:         return try{
17:             connection = (URL(params[0])?.openConnection() as?
                HttpsURLConnection)
18:             connection?.requestMethod = "POST"
19:             connection?.doOutput = true
20:             connection?.doInput = true
21:             connection?.setRequestProperty("Content-Type","application/
                json")
22:             val message : JSONObject = JSONObject()
23:             message.put("code",params[1])
24:             val outputWriter = OutputStreamWriter(connection?.
                outputStream)
25:             outputWriter.write(message.toString())
26:             outputWriter.flush()
```

```
27:             if (connection?.responseCode == 200){
28:                 val inputStream = InputStreamReader(connection?.
                    inputStream)
29:                 val body = JSONObject(inputStream.readText())
30:                 return body.getString("message")
31:             } else{
32:                 return "error"
33:             }
34:         } finally {
35:             connection?.disconnect()
36:         }
37:     }
38:
39:     override fun onPostExecute(result: String){
40:         mActivity.setText(result)
41:     }
42: }
```

You will also need to add the following permissions (in bold) to your AndroidManifest.xml file in the `manifest` section:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.redteamlife.aas2.aas2client">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

After building and running, you should be able to interact with it and see either error or success messages being displayed like in Figure 9-7.

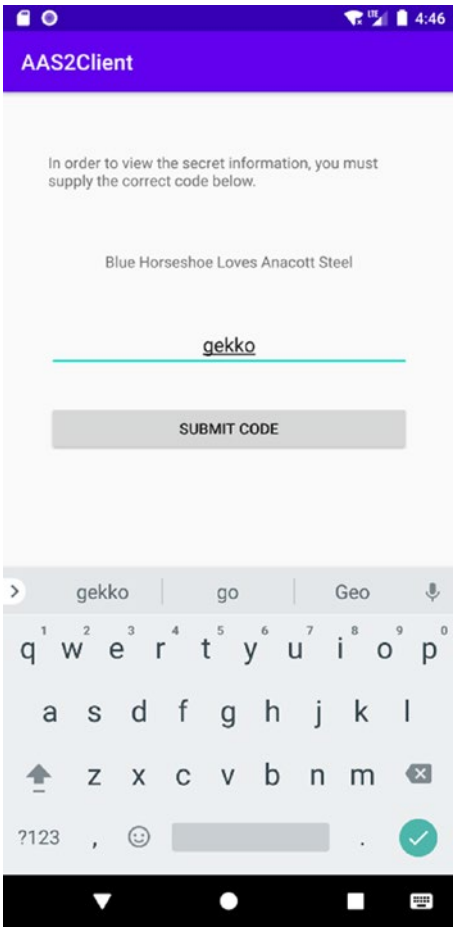


Figure 9-7. *A success message after entering the correct code*

Testing SSL Traffic Interception with Burp Suite

We are now ready to see if we can intercept traffic from between our client app and our back-end server. First let's fire up Burp Suite and then install its certificate on our Android device. For this demo, I'm using the emulator. Go into the Proxy upper tab, then Options lower tab. Then select Import/export CA certificate, and in the resulting dialog, select Export ➤ Certificate in DER format as shown in Figure 9-8. Then select a location, name the file, and save it.

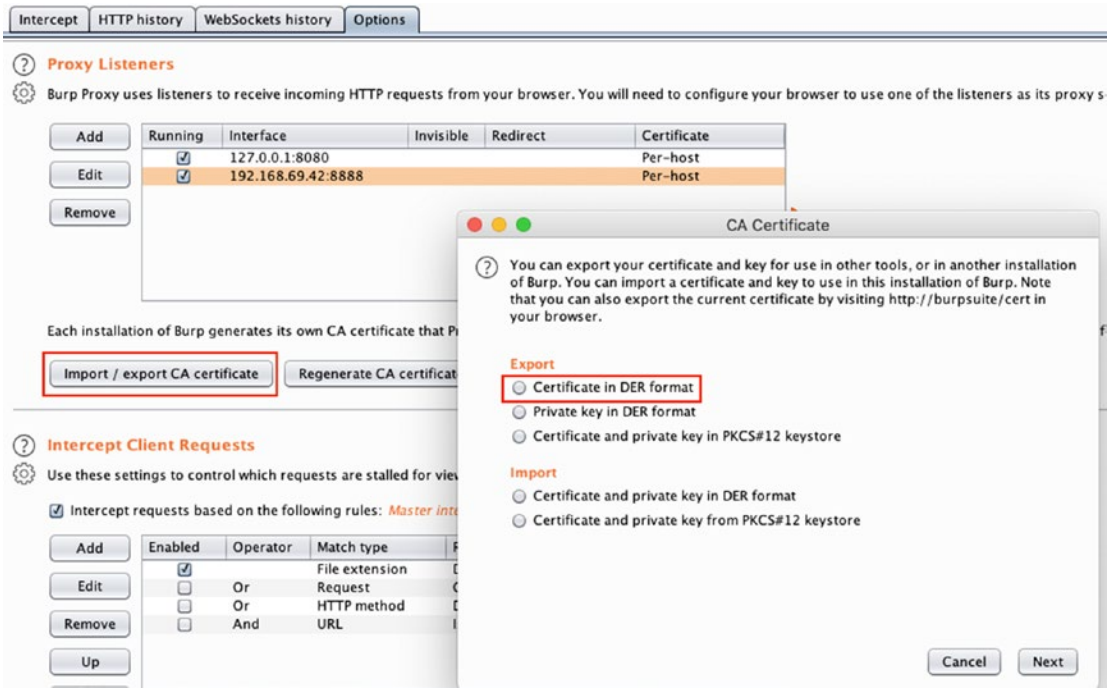


Figure 9-8. *Exporting the Burp CA certificate*

Next, let's copy that certificate file over to our Android device:

```
➜  adb push burpcert.cer /data/media/0/Download/
burpcert.cer: 1 file pushed, 0 skipped. 0.2 MB/s (973 bytes in 0.004s)
➜
```

Now we have to install it on our Android device. So, on the device, go to Settings ➤ Security and Location ➤ Encryption & Credentials ➤ Install From SD Card. In the file browser that pops up, navigate to your Download folder and select the certificate as shown in Figure 9-9 and then proceed to name it and install it as shown in Figure 9-10.

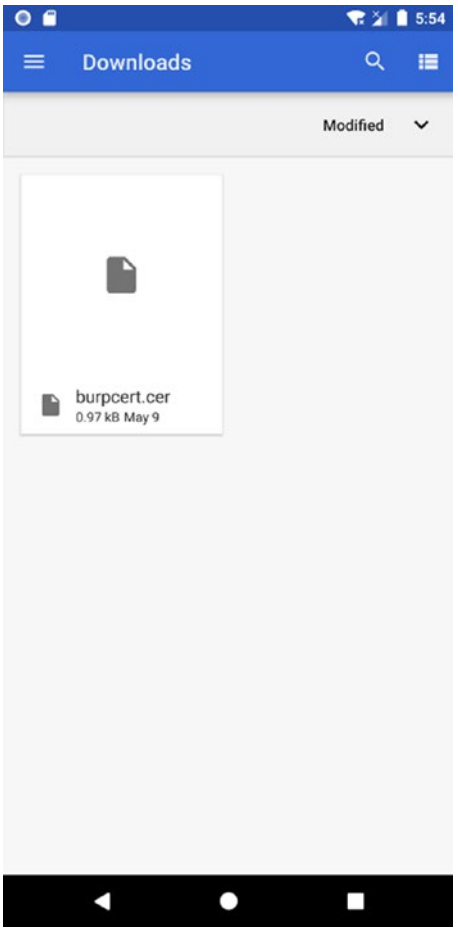


Figure 9-9. *Burp Suite CA certificate that we copied to the Downloads directory*

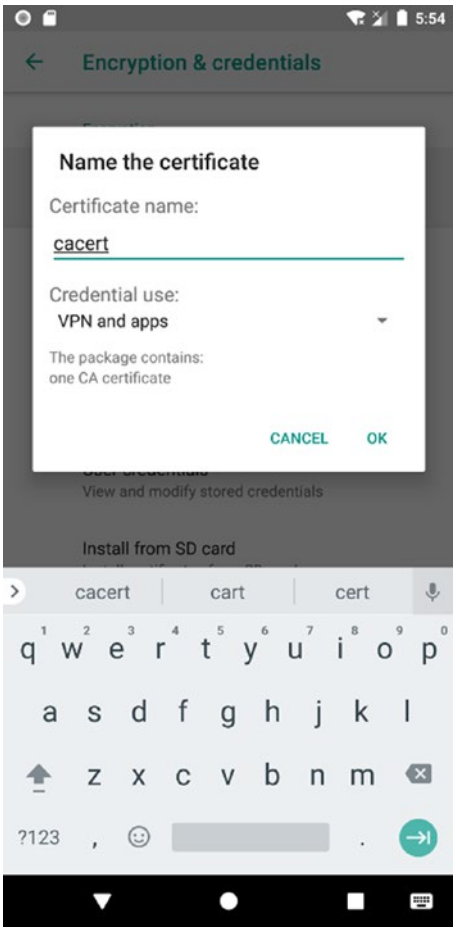


Figure 9-10. *Installing the Burp CA certificate*

Doing this will tell the device to trust any server certificates that are signed by the Burp Suite CA certificate. Now, we have to route all our traffic from the device to Burp Suite so it can relay that data to the server and effectively become our man in the middle. To do this, go to Settings ➤ Network & Internet ➤ Wifi and then select AndroidWifi (if you're using an emulator) or the network that you are currently connected to. Make sure that both your Android device and Burp Suite are connected to the same network so that they can easily send traffic back and forth. Remove any firewalls between the two devices if any, for example, if you have any antivirus software or something like either Little Snitch for MacOS X or iptables on Linux. Modify this network as shown in Figure 9-11, and in Proxy, select Manual and then fill in the IP address and port of your Burp Proxy. In my case that is 192.168.69.42:8888.

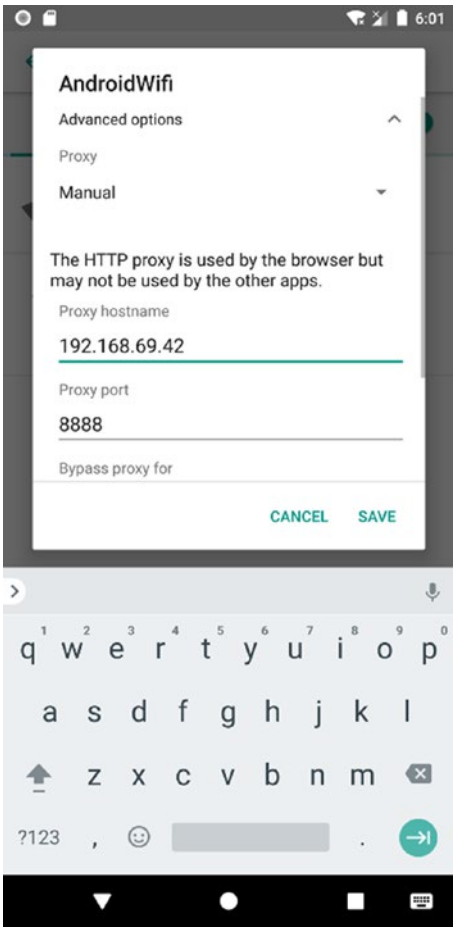


Figure 9-11. *Adding Burp Proxy to the device so that all traffic is routed via the proxy*

One more thing, switch off your mobile data on your Android device. This is because Android will use this as a backup in case it detects something unusual with the Wi-Fi connection, for example, it detects a fake certificate. OK, I think we're ready to test SSL traffic interception! Fire up the client app and enter some codes and click submit.

Two things may happen at this point. First, you will see the SSL traffic in Burp's Proxy section. Or second, the app will crash. When I first did it, the app crashed on me. So I did a little further digging, and I found that since Android version 7 (API 24), the engineers over at Google have reconfigured how trusted Certificate Authorities handle CA certificates. There is a link to a blog post here: `https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html`. What does this mean for us? Well, it means that even if we add user CAs (like we did) to our Android

device, an app has to specifically opt in to work with that user CA certificate – the premise being that if we were able to add it to the device, we should also be able to add it to the network security config. Turns out, though, that we actually can! But we need to play around with the APK first a little. Let's do that now.

Pull out the APK like you learned to in Chapter 5. Then using APKTool (also mentioned in that chapter), disassemble the APK. You then have to do two things. First, you have to create a file called network_security_config.xml inside the <app>/res/xml/ directory. Put the following declarations in the file:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
      <base-config>
           <trust-anchors>
                <certificates src="user" />
           </trust-anchors>
       </base-config>
 </network-security-config>
```

Then, you have to edit the AndroidManifest.xml file to include the file you just created. Add the parts in bold to your AndroidManifest.xml file:

```
<application android:allowBackup="true" android:appComponentFactory=
"androidx.core.app.CoreComponentFactory" android:debuggable="true"
android:extractNativeLibs="false" android:icon="@mipmap/ic_launcher"
android:label="@string/app_name" android:roundIcon="@mipmap/ic_launcher_
round" android:supportsRtl="true" android:testOnly="true" android:theme=
"@style/AppTheme" android:networkSecurityConfig="@xml/network_security_
config">
```

Then repack and sign your APK once again. I like to rename the APK that I rebuild just to differentiate it from the original. Here's what it looks like on my workstation:

```
➜  apktool b -o aas2_nsc.apk aas2client/
I: Using Apktool 2.4.0
I: Checking whether sources has changed...
I: Checking whether sources has changed...
I: Checking whether resources has changed...
I: Building apk file...
```

```
I: Copying unknown files/dir...
I: Built apk...
➜  apksigner sign --ks /Users/sheran/keystore.ks aas2_nsc.apk
Keystore password for signer #1:
➜
```

Lastly, install it on the device and test again. This time, it should work just fine, and you will be able to see SSL traffic flowing back and forth as shown in Figures 9-12 and 9-13. Now keep in mind, this is for an app that hasn't had SSL Pinning implemented. So, let's build in some SSL Pinning into our app.

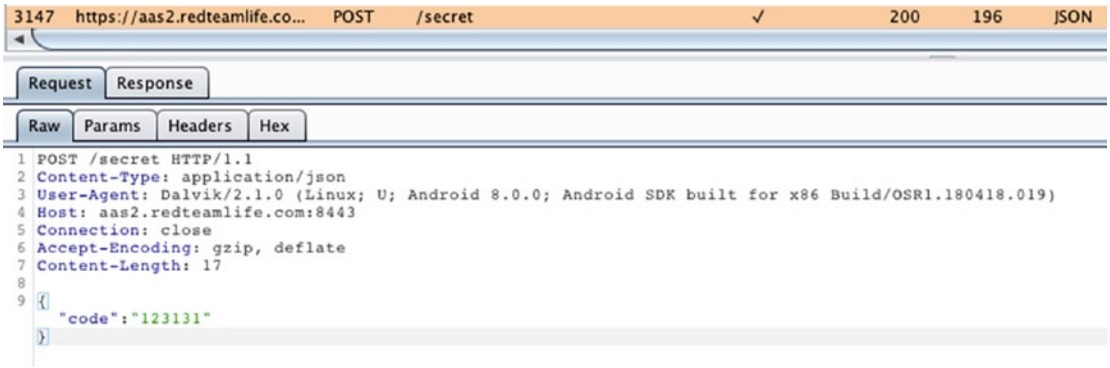


Figure 9-12. *The intercepted SSL Request*

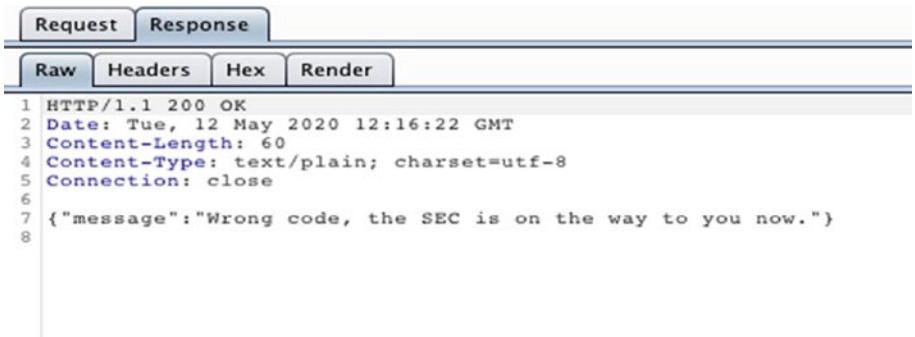


Figure 9-13. *The intercepted SSL Response*

Adding SSL Pinning

You can use SSL Pinning in two ways; one is through the Network Security Configuration. To pin a certificate using the network security config, you have to create an XML file similar to what we did, but instead containing a hash of the public key you want to pin to. To generate that hash, you can use this string of OpenSSL commands:

```
➜   openssl x509 -in fullchain.pem -noout -pubkey |openssl pkey -pubin
    -outform der |openssl dgst -sha256 -binary|openssl enc -base64
    jM2RG/WsDtG849S7Inoq7tc3O1pyWewWIlH7lFyfrVc=
➜
```

Once you have this hash, you can place it in your network_security_config.xml file like so:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">aas2.redteamlife.com</domain>
            <pin-set>
            <pin digest="SHA-256">jM2RG/WsDtG849S7Inoq7tc3O1pyWewWIlH7lFyf
            rVc=</pin>
            </pin-set>
    </domain-config>
</network-security-config>
```

If you put that into your app and repackage it with APKTool and test it, your app should crash.

An alternate way, the older way, to do SSL Pinning was using code and the TrustManager. Let's take a look at an example. Here, I will be modifying our existing client's `NetUtils.kt` file and adding our PEM certificate to the `/res/raw` directory. Code follows:

```
01: package com.redteamlife.aas2.aas2client
02:
03: import android.os.AsyncTask
04: import android.util.Log
05:
```

```
06: import org.json.JSONObject
07: import java.io.*
08: import java.net.URL
09: import java.security.KeyStore
10: import java.security.cert.CertificateFactory
11: import javax.net.ssl.*
12: import java.security.cert.X509Certificate
13:
14: class NetworkAsyncTask(activity: MainActivity): AsyncTask<String, Void,
    String>(){
15:
16:     private val mActivity = activity
17:
18:     override fun doInBackground(vararg params: String?): String? {
19:         val cf: CertificateFactory = CertificateFactory.
          getInstance("X.509")
20:
21:         val caInput: InputStream = BufferedInputStream(mActivity.
          resources.openRawResource(R.raw.cert))
22:         val ca: X509Certificate = caInput.use {
23:             cf.generateCertificate(it) as X509Certificate
24:         }
25:
26:         val keyStoreType = KeyStore.getDefaultType()
27:         val keyStore = KeyStore.getInstance(keyStoreType).apply {
28:             load(null, null)
29:             setCertificateEntry("ca", ca)
30:         }
31:
32:         val tmfAlgorithm: String = TrustManagerFactory.
          getDefaultAlgorithm()
33:         val tmf: TrustManagerFactory = TrustManagerFactory.
          getInstance(tmfAlgorithm).apply {
34:             init(keyStore)
35:         }
36:
```

```
37:         val context: SSLContext = SSLContext.getInstance("TLS").apply {
38:             init(null, tmf.trustManagers, null)
39:         }
40:
41:         var connection: HttpsURLConnection? = null
42:         return try{
43:             connection = (URL(params[0])?.openConnection() as?
                HttpsURLConnection)
44:             connection?.requestMethod = "POST"
45:             connection?.doOutput = true
46:             connection?.doInput = true
47:             connection?.setRequestProperty("Content-Type","application/
                json")
48:             connection?.sslSocketFactory = context.socketFactory
49:             connection?.connect()
50:
51:             val message : JSONObject = JSONObject()
52:             message.put("code",params[1])
53:             val outputWriter = OutputStreamWriter(connection?.
                outputStream)
54:             outputWriter.write(message.toString())
55:             outputWriter.flush()
56:             if (connection?.responseCode == 200){
57:                 val inputStream = InputStreamReader(connection?.
                    inputStream)
58:                 val body = JSONObject(inputStream.readText())
59:                 return body.getString("message")
60:             } else{
61:                 return "error"
62:             }
63:         } finally {
64:             connection?.disconnect()
65:         }
66:     }
67:
```

```
68:     override fun onPostExecute(result: String){
69:         mActivity.setText(result)
70:     }
71:
72:
73: }
```

Lines 19–39 and 48 are different as you can see in the code in bold. What we're doing here is storing the certificate from our server aas2.redteamlife.com in the app. Then, we read it and add it to a KeyStore and then use that KeyStore to initialize a TrustManager which we then pass on to an SSLContext. Then, the SSLContext is passed to our HttpsURLConnection. In this way, the TrustManager will ensure that it can verify all certificates against the one we included and pinned. Now when I test the new client, my app crashes right away. There are a few points here to keep in mind. First is on the pinning side. Generally, it is a better idea to pin a public key as opposed to an actual certificate like we have done. This is because when a certificate gets renewed, its public key still remains the same. Thus, each time we renew a certificate, it wouldn't mean we have to make a code change on our client. So it's always better to pin a public key. I will show you that later in the chapter. The next point to consider is the continuation. You have a choice with what to do when your certificate pinning validation fails. If the exception is thrown, then you know that something fishy is going on and at that point can take action. Would you continue with lesser features or would you completely stop? In my opinion, it is better to completely stop communications until the certificate issue has been resolved. In our case, we're failing fairly inelegantly. You could choose to notify the user that there's an issue with the server certificate or even just fail with a generic error message. With that said, let's move on to the real reason you're here, breaking SSL Pinning.

Breaking SSL Pinning

The TrustManager, as its name implies, plays a pivotal role in ensuring that server certificates are trusted. If you look at the TrustManager class or rather the X509TrustManager, which is the subclass most often used, you can see three public methods: `checkClientTrusted()`, `checkServerTrusted()`, and `getAcceptedIssuers()`. We are most interested in the `checkServerTrusted()` method which we want to render nonfunctional. There are possibly many ways to do this, but hooking this function

and rewriting it is probably the cleanest way. We would use a tool like Frida in this case to load a script that has our own implementation of the X509TrustManager. checkServerTrusted() method. But we can take a shorter cut. Essentially, the easiest way is to create our own TrustManager which is empty and then send that into the SSLContext's init method by hooking it. Since our TrustManager has no actual checks in place, we should be able to get past the SSL Pinning in this case. So, let's try that out.

First, let's create a script in our Frida directory. I'm calling mine trustmgr.js. In Frida's JavaScript format, that would look something a little like this:

```
const TrustManager = Java.registerClass({
                name : 'com.redteamlife.aas2.X509TrustManager',
                implements : [X509TrustManager],
                methods : {
                    checkClientTrusted: function (chain, authType) {},
           checkServerTrusted: function (chain, authType) {},
           getAcceptedIssuers: function () {return []; },
            }
       })
```

You can see clearly it has zero functionality other than just serving as empty placeholders for each of the methods. Now, let's put that into a full-fledged script like so:

```
01: setTimeout(function() {
02:     Java.perform(function () {
03:          const X509TrustManager = Java.use('javax.net.ssl.
             X509TrustManager');
04:          const SSLContext = Java.use('javax.net.ssl.SSLContext');
05:
06:          const TrustManager = Java.registerClass({
07:               name : 'com.redteamlife.aas2.X509TrustManager',
08:               implements : [X509TrustManager],
09:               methods : {
10:                 checkClientTrusted: function (chain, authType) {},
11:                 checkServerTrusted: function (chain, authType) {},
12:                 getAcceptedIssuers: function () {return []; },
13:               }
14:          })
```

```
15:          var TrustManagers = [TrustManager.$new()];
16:         try {
17:             SSLContext.init.overload(
18:             '[Ljavax.net.ssl.KeyManager;', '[Ljavax.net.ssl.
                TrustManager;', 'java.security.SecureRandom').
                implementation = function(keyManager, trustManager,
                secureRandom) {
19:                 SSLContext.init.call(this, keyManager, TrustManagers,
                    secureRandom);
20:             };
21:         } catch (err) {
22:             console.log(err);
23:         }
24:     });
25: });
```

Line 17, while a bit of an eyeful, is where the magic happens. Here we are calling SSLContext.init with an empty TrustManager. Now if you recall, in our client, we went to great pains to build our TrustManager with a KeyStore and our pinned certificate. Here, we're completely negating that piece of code. So instead of the SSLContext init taking place as per our client code, Frida overwrites and calls the init function with our new, empty TrustManager. All that's left now is to run Frida with our script as follows:

```
(p3) ➜  frida -U -f com.redteamlife.aas2.aas2client -l trustmgr.js --no-pause

     ____
    / _  |   Frida 12.8.20 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://www.frida.re/docs/home/
Spawned `com.redteamlife.aas2.aas2client`. Resuming main thread!
[Android Emulator 5554::com.redteamlife.aas2.aas2client]->
```

We get dropped into the Frida shell, but we will notice that our emulator just started up our app. So let's go ahead and test it now. Bam! SSL Pinning is broken on my emulator. A quick peek at Burp Suite verifies this as I see the data going in between my client and server (Figure 9-14).

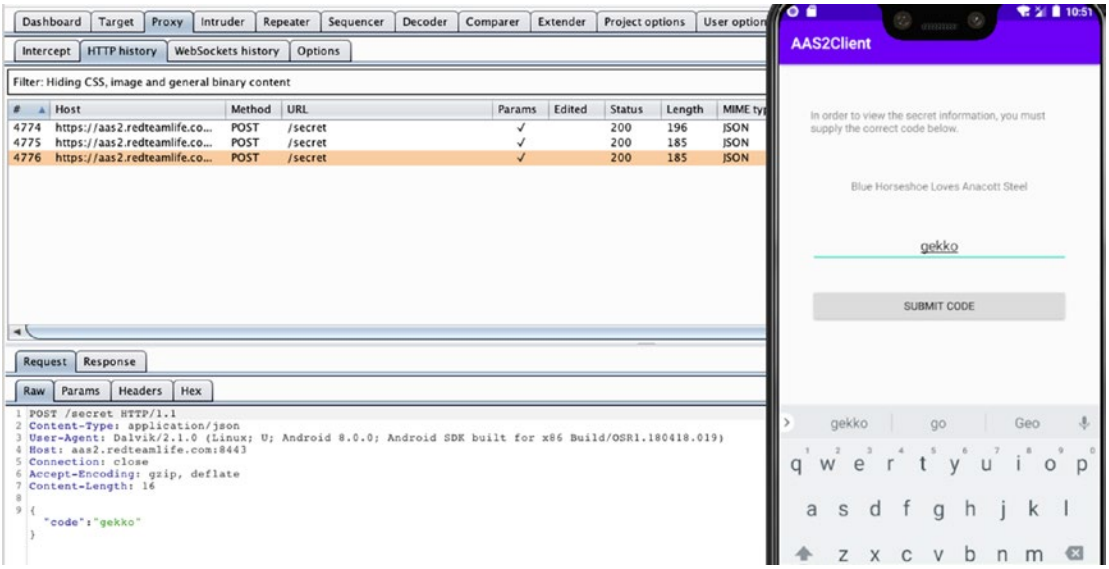


Figure 9-14. *The data going in between client and server intercepted by Burp*

I would like to quickly revisit the topic of the way Android 7 and above handled Certificate Authorities. If you recall, we used Network Security Configuration to trust all our user certificates. There is another way, however – again, by using Frida. But to do that, we have to dig into the Android source a little bit. Google decided to fork OpenSSL and write their own library called BoringSSL [https://boringssl.googlesource.com/boringssl/]. They use the BoringSSL library in their own projects Google Chrome and notably Android. Android's implementation of BoringSSL is through Conscrypt [https://source.android.com/devices/architecture/modular-system/conscrypt]. Conscrypt is a Java Security Provider used in Android together with the BoringSSL native library to provide Android's cryptographic functionality such as ciphers, key generation, and message digests. It also provides Android's TLS implementation. Since there are many common functions with OpenSSL, let's take a quick look at how OpenSSL does its certificate verification.

OpenSSL allows you to set a verification callback function from the functions `SSL_CTX_set_verify` or `SSL_set_verify`. The difference being the former works for all `ssl`

objects derived from a single context, `ctx`, and the latter works only on the `ssl` object that it is called on. What both functions do is call your user-defined verification callback function when it is time to verify a peer certificate. You can find out more about these function calls on the OpenSSL website here: www.openssl.org/docs/manmaster/man3/SSL_CTX_set_verify.html. I am bringing this up because BoringSSL and Conscrypt make use of a similar function. If you look at the embedded Android Conscrypt code, especially in the class `com.android.org.conscrypt.NativeCrypto`, you will see a similar `SSL_set_verify` method which calls the underlying native BoringSSL library. Figure 9-15 shows how Android Oreo handles its certificate verification.

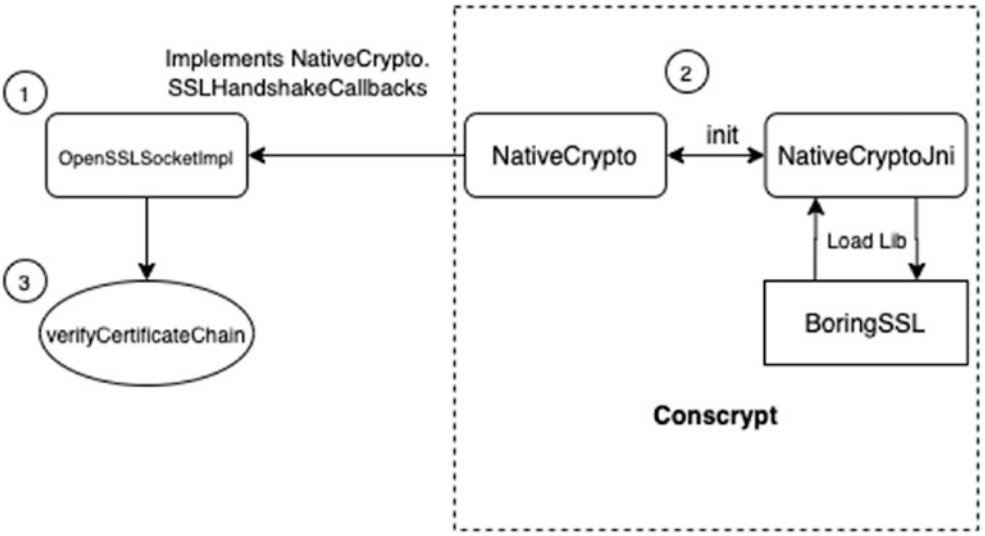


Figure 9-15. *How Android uses Conscrypt to verify certificates*

First (point 1), the OpenSSLSocketImpl class will instantiate the NativeCrypto class which contains the relevant callbacks a little similar to how OpenSSL is. NativeCrypto (point 2) will then load the native library through NativeCryptoJni which interestingly looks in several different places for the library as shown in the code here:

```
01: package org.conscrypt;
02: /**
03:  * Helper to initialize the JNI libraries. This version runs when
    compiled as part of an app distribution (or GmsCore).
04:  *
05:  */
```

```
06: class NativeCryptoJni {
07:     public static void init() {
08:         if ("com.google.android.gms.org.conscrypt".equals(NativeCrypto.
            class.getPackage().getName())) {
09:             System.loadLibrary("gmscore");
10:             System.loadLibrary("conscrypt_gmscore_jni");
11:         } else {
12:             System.loadLibrary("conscrypt_jni");
13:         }
14:     }
15:     private NativeCryptoJni() {
16:     }
17: }
```

Lastly (point 3), OpenSSLSocketImpl will implement the verifyCertificateChain method which is the callback method that is called by Conscrypt during the SSL connection setup. Why am I telling you all this? Well, if we overwrite the verifyCertificateChain method using Frida with this line of code:

```
1: var OpenSSLEngineSocketImpl = Java.use('com.android.org.conscrypt.
   OpenSSLEngineSocketImpl');
2: OpenSSLSocketImpl.verifyCertificateChain.overload('[Ljava.lang.Long;',
   'java.lang.String').implementation = function(certRefs, authMethod) {};
```

then we can easily avoid the pesky issue of having to add a new network_security_config.xml file that asked Android to trust all user-added certificates.

Other Pinning Techniques

There are quite a few SSL Pinning techniques that I am not going to cover here. I will list down some of the ones that I am aware of, however:

- OkHTTPv3 pinner
- Trustkit pinner
- Appcelerator PinningTrustManager pinner

- OpenSSLEngineSocketImpl Conscrypt pinner
- OpenSSLSocketImpl Apache Harmony pinner
- PhoneGap sslCertificateChecker pinner
- IBM MobileFirst pinTrustedCertificatePublicKey pinner
- IBM WorkLight HostNameVerifierWithCertificatePinning pinner
- Conscrypt CertPinManager pinner
- CWAC-Netsecurity CertPinManager pinner
- Worklight Androidgap WLCertificatePinningPlugin pinner
- Netty FingerprintTrustManagerFactory pinner
- Squareup CertificatePinner pinner
- Squareup OkHostnameVerifier pinner
- Apache Cordova WebViewClient pinner
- Boye AbstractVerifier pinner

This list comes from the Frida codeshare script `frida-multiple-unpinning.js` found here: https://codeshare.frida.re/@akabe1/frida-multiple-unpinning/. Frida's codeshare section is an open platform where users can submit their own scripts. Then by using Frida's "—codeshare" switch, you can use these scripts without having to download them. If you look further on codeshare, you can see a whole slew of scripts dedicated to removing SSL Pinning. It is important to take note that these scripts usually show up when a new pinning mechanism has been found. That goes to show you that in general, reverse engineers are always going to come out on top. Tools such as obfuscation, moving code to native libraries, and such can act as a deterrent and possibly slow down a reverse engineer. But if there is determination and time, in the end, your app is bound to succumb to the attacks that are thrown at it. This isn't meant to discourage you but serve as a warning that it is best to not expose as much code or data as possible and hope that the security tools solve the problem.

Now, I'll show you the technique of how to pin a certificate's public key vs. the certificate. We make no other changes to our client app except for in NetUtils.kt as follows:

```
001: package com.redteamlife.aas2.aas2client
002:
003: import android.net.http.X509TrustManagerExtensions
004: import android.os.AsyncTask
005:
006: import android.util.Base64
007: import android.util.Log
008:
009:
010: import org.json.JSONObject
011: import java.io.*
012: import java.net.URL
013: import java.security.KeyStore
014: import java.security.MessageDigest
015: import java.security.NoSuchAlgorithmException
016: import java.security.cert.Certificate
017: import java.security.cert.X509Certificate
018: import java.util.*
019: import javax.net.ssl.*
020: import javax.security.cert.CertificateException
021:
022: public class NetworkAsyncTask(activity: MainActivity):
     AsyncTask<String, Void, String>(){
023:
024:     private val mActivity = activity
025:
026:     override fun doInBackground(vararg params: String?): String? {
027:         val trustManagerFactory: TrustManagerFactory =
             TrustManagerFactory.getInstance(TrustManagerFactory.
             getDefaultAlgorithm())
028:         val ks: KeyStore? = null
029:         trustManagerFactory.init(ks)
030:         var x509TrustManager: X509TrustManager? = null
```

```
031:        for(trustManager: TrustManager in trustManagerFactory.
            trustManagers) run lit@{
032:            if ( trustManager is  X509TrustManager){
033:                x509TrustManager = trustManager
034:                return@lit
035:            }
036:        }
037:        val trustManagerExt = X509TrustManagerExtensions(x509Trust
            Manager)
038:
039:        var connection: HttpsURLConnection? = null
040:        return try{
041:            connection = (URL(params[0])?.openConnection() as?
                HttpsURLConnection)
042:            connection?.requestMethod = "POST"
043:            connection?.doOutput = true
044:            connection?.doInput = true
045:            connection?.setRequestProperty("Content-
                Type","application/json")
046:            connection?.connect()
047:            val validPins: Set<String> = Collections.singleton("jM2RG/
                WsDtG849S7Inoq7tc3O1pyWewWIlH7lFyfrVc=")
048:            validatePinning(trustManagerExt,connection,validPins)
049:
050:            val message : JSONObject = JSONObject()
051:            message.put("code",params[1])
052:            val outputWriter = OutputStreamWriter(connection?.
                outputStream)
053:            outputWriter.write(message.toString())
054:            outputWriter.flush()
055:            if (connection?.responseCode == 200){
056:                val inputStream = InputStreamReader(connection?.
                    inputStream)
057:                val body = JSONObject(inputStream.readText())
058:                return body.getString("message")
```

```
059:             } else{
060:                 return "error"
061:             }
062:         } finally {
063:             connection?.disconnect()
064:         }
065:     }
066:
067:     override fun onPostExecute(result: String){
068:         mActivity.setText(result)
069:     }
070:
071:     @Throws(SSLException::class)
072:     private fun validatePinning(trustManagerExt:
         X509TrustManagerExtensions, conn: HttpsURLConnection?, validPins:
         Set<String>)  {
073:         var certChainMsg: String = ""
074:         try{
075:             val md: MessageDigest = MessageDigest.
                 getInstance("SHA-256")
076:             val trustedChain: List<X509Certificate> = trustedChain
                 (trustManagerExt,conn)
077:             for( cert: X509Certificate in trustedChain ) run {
078:                 val publicKey: ByteArray = cert.publicKey.encoded;
079:                 md.update(publicKey,0,publicKey.size)
080:
081:                 var pin: String = Base64.encodeToString(md.
                     digest(),Base64.NO_WRAP)
082:                 certChainMsg += "    sha256/" + pin + " : " + cert.
                     subjectDN.toString() + "\n"
083:                 if (validPins.contains(pin)){
084:                     return;
085:                 }
086:             }
087:         } catch(e: NoSuchAlgorithmException){
```

```
088:             throw SSLException(e)
089:         }
090:         throw SSLPeerUnverifiedException("Pinning Fail! Chain:
             \n"+certChainMsg)
091:     }
092:
093:     @Throws(SSLException::class)
094:     private fun trustedChain(trustManagerExt:
         X509TrustManagerExtensions, conn: HttpsURLConnection?):
         List<X509Certificate>{
095:         val serverCerts: Array<Certificate> = conn?.
             serverCertificates!!
096:         val untrustedCerts: Array<X509Certificate> = serverCerts.map {
             it as X509Certificate }.toTypedArray()
097:         val host: String = conn?.url!!.host
098:         try{
099:             return trustManagerExt.checkServerTrusted(untrustedCerts,
                 "RSA", host)
100:         } catch(e: CertificateException) {
101:             throw SSLException(e)
102:         }
103:     }
104: }
```

I've made the changes bold so you can clearly see what we are doing. We have two new functions called `trustedChain` and `validatePinning`. `trustedChain` will pick up all the certificates from the server and then run it through its own X509TrustManagerExtensions check. The difference between the X509TrustManagerExtensions and plain X509TrustManager is that the checkServerTrusted method will verify and also return the list of certificates to the caller. If it cannot, then the method will throw an SSLException. If it can verify, then it gets verified by the validatePinning method to see if one of the certificates has a matching public key hash that we compare with. If this also fails, then again an SSLException is thrown. If it passes, the connection proceeds as planned.

The preceding piece of code isn't mine, but I did translate it into Kotlin from Java. The code itself was featured in a blog post by Matthew Dolan here: https://medium.com/@appmattus/android-security-ssl-pinning-1db8acb6621e. In this post, he also provides a few other mechanisms of SSL Pinning. It is highly worth a read. If you notice this code, it doesn't actually conform to any of the present SSL pin-breaking mechanisms we have with Frida. I am not a fan of security through obscurity, but if you wanted to further hide what this class was doing, you could rename the methods and fields, but a reverser will always be able to find it through the exceptions. You could then possibly roll your own exceptions, but again, it's a case of following the breadcrumbs. Lastly, you could obfuscate all of that, but then on a rooted phone, a reverser could trace all the native calls and work backward from there. As discouraging as this sounds, only the most determined reversers with something to gain will go after you. Usually, the current mechanisms for protecting apps in the form of obfuscation are sufficient for the drive-by hackers. You can find the TrustManager pinning client source code here: https://github.com/sheran/aas2-ch09-android_client_2.

Summary

In summary, it is important to keep in mind that your app is effectively a showcase for your code and data. It's just a more structured way of sending out your intellectual property to the world. Your app will be installed on many mobile devices, and a percentage of those will very likely be hostile. Your goal should be to first limit the amount of exposure by reducing what you release in your app, whether it is code or data. Then, if there are parts that you must lock down with obfuscation, do so. SSL is nonnegotiable of course. Always implement SSL Pinning so that even a drive-by hacker can't see client and server data exchange. But always be prepared that they will be able to see this data and take steps accordingly. Like I said in my first edition, a healthy dose of paranoia when developing your app will serve you well.

CHAPTER 10

Looking Ahead

So far, what we've seen is bleak. I intended it that way on purpose. I do celebrate the reverse engineers and people that pick apart apps. I do, however, want you to know that you are in a difficult position and can't afford to take shortcuts or be lazy about security. My goal of showing you and possibly other reverse engineers how to break Android apps was to get you to take a more serious approach to your security. You have to learn about the principles and work from there rather than rely on someone to build you yet another framework to help simplify your life. Therefore, in this chapter, I want to take you through some alternatives in security. These alternatives are still new and thus have not had enough of an impact on the reversing community as a whole. To this end, you will not find many blog posts or articles written about these topics. I think that you owe it to yourself to figure out some new ways of protecting your own apps, though. It doesn't hurt and can only help strengthen the ecosystem further.

What I will do in this chapter is talk about some alternate approaches to handling sensitive aspects of app development – SSL Pinning being one of them. We will take a look at how some frameworks are designed and work and then consider alternatives to doing SSL Pinning based on the fact that Android is still, essentially, Linux. These topics are new even to me, and as such I am actively researching them and will publish more details on the companion website for this book [`https://aas2book.com`]. I would love to work closer with you if you have any thoughts on developing what I present here or even contributing to it. Thus, without much delay, let's get started.

Flutter

Flutter is an app development framework by Google, very similar to React and React Native. You can find out more on its website: `https://flutter.dev`. Flutter revolves around Google's answer to JavaScript called Dart. The Dart programming language has been around since 2011, and if you aren't aware of it yet, then take a closer look here:

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8_10

https://dart.dev. What's interesting with Dart is that it can compile to native code or to JavaScript. The native side of things is what should appeal to us as mobile developers. By compiling into native code, Dart and by extension Flutter can be used on multiple supported platforms. An overview of the Flutter system is shown in Figure 10-1.

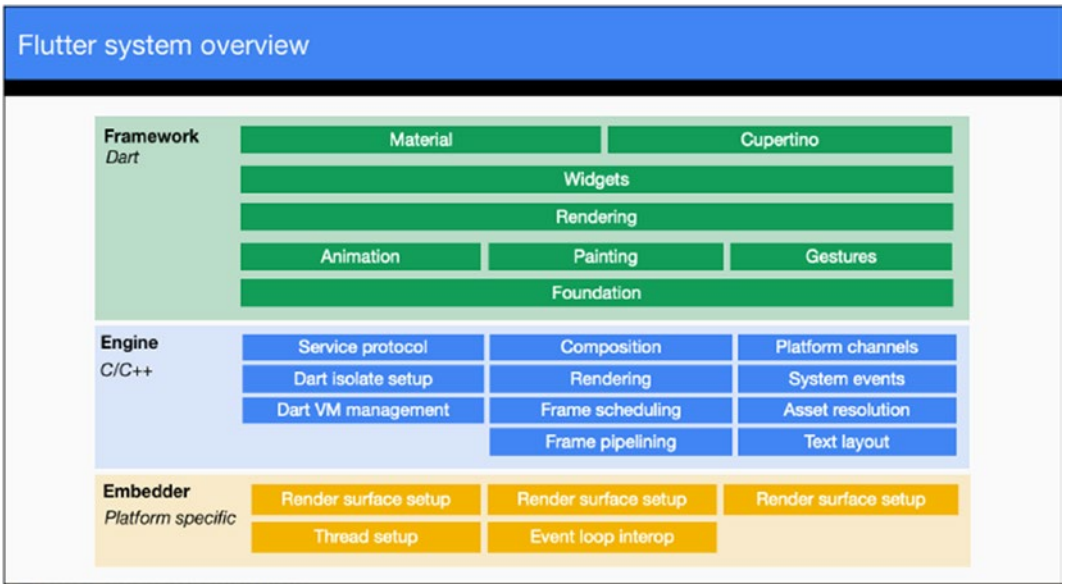


Figure 10-1. *Flutter system overview taken from https://flutter.dev/docs/resources/technical-overview*

By changing just the Embedder layer, it is possible to transplant Flutter apps to both iOS and Android. There is even a web component that is equivalent to building HTML5 apps through other frameworks like Titanium, Ionic, Apache Cordova, and so on. There, the premise is to use HTML5 and JavaScript to build apps which compile to native packages. This is similar except that Flutter uses Dart, and the Flutter ethos is that you build, use, and reuse Widgets.

I won't be covering Flutter in depth in this chapter and rely on you to do some of your own research. Flutter has some great installation instructions on how to get setup for the first time, so I encourage you to follow those instructions and get your developer environment setup. Start here [https://flutter.dev/docs/get-started/install] and continue on to the Test Drive section here [https://flutter.dev/docs/get-started/test-drive]. This is the standard demo that we will begin to work off of. My goal in this section is to show you an alternative to using SSL Pinning that is a little trickier to defeat.

Fair warning, I have managed to break SSL Pinning described in this section and still intercept traffic going in between, but it was not straightforward and required some considerable time and research.

Flutter is interesting in that it uses its own HTTP communication mechanisms (from Dart) and comes bundled with its own set of trusted root certificates. What does this mean when you want to intercept SSL traffic from a Flutter app? Well, it means that even if you set the phone's proxy to point to your Burp Proxy location and even if you add the Burp Suite CA certificate, you will not be able to see the Flutter app's traffic passing through the proxy. I've outlined what that looks like in Figure 10-2.

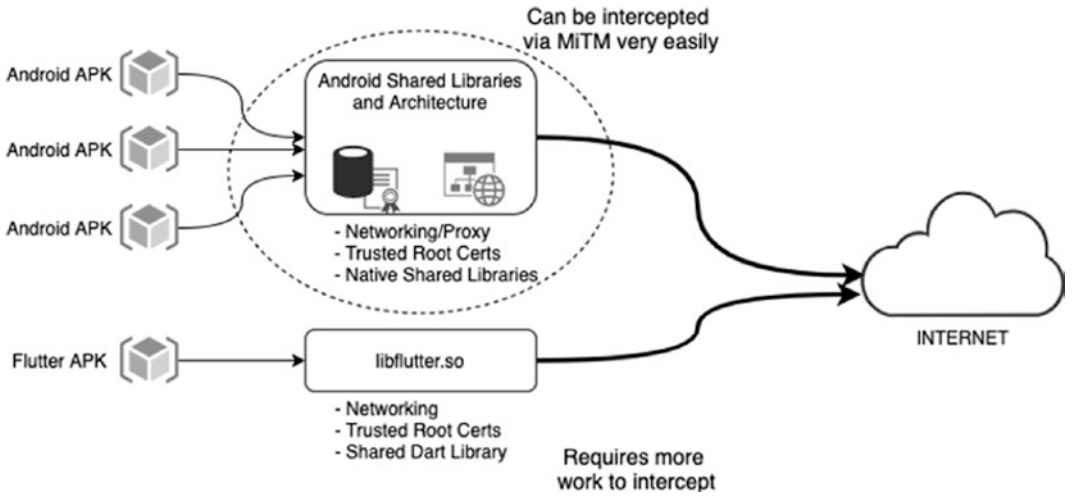


Figure 10-2. *Flutter maintains its own networking and trusted root CAs*

Flutter apps will have their own networking and trusted root CAs which is why they are not affected by the overall Android system settings. Therefore, to fully defeat SSL certificate verification, it is a little bit more of an elaborate process where you must look at the libflutter.so native library within the APK to find out where the SSL certificate verification takes place and then render it harmless through our standard techniques of using Frida. To divert all traffic, you will have to use something like iptables to route all network traffic to an IP address (your proxy server). SSL traffic can be looked at, but it isn't as straightforward as how things are usually done.

The Flutter Certificate Verification

So how would this look in practice? Let's see. In your project that you created, locate the main.dart file. It will be in the /lib directory. Open the file and look for the _ incrementCounter() function. It should be around line 55. You will see that the function looks like this:

```
void _incrementCounter() {
    setState(() {
      // This call to setState tells the Flutter framework that something has
      // changed in this State, which causes it to rerun the build
      method below
      // so that the display can reflect the updated values. If we changed
      // _counter without calling setState(), then the build method would
      not be
      // called again, and so nothing would appear to happen.
      _counter++;
    });
  }
```

What we're going to do is add our code to this function. Now normally, you would write all these widgets yourself, but the way we structure it, we're going to make an HTTPS call whenever the button on this demo app is pressed. First, we have to add the http package. To do this, open your pubspec.yaml file, and under the dependencies, add this line:

```
dependencies:
  http: 0.12.1
```

Then add this line to your main.dart imports section up top:

```
import 'package:http/http.dart' as http;
```

Next, we will make an HTTPS request to Google. Add this line to your _ incrementCounter function (I also removed the comments):

```
void _incrementCounter() {
    http.get("https://google.com").then((response) => print(response.
    headers));
```

```
    setState(() {
      _counter++;
    });
  }
```

With this extremely contrived example, what we do is make a request over SSL to google.com and fetch the response headers and print them out. I am not printing the response body because Flutter is limited by Android in how much data it can write to logcat. When you run your app, you should see a screen that looks like the one in Figure 10-3 either on your emulator or device.

Figure 10-3. *Flutter demo app*

Now when you click the + button, your request is made to google.com and the output printed out into logcat. As an exercise, try intercepting traffic of this app the way we did in Chapter 9. You will find that you won't see any traffic or notice how Burp Proxy doesn't complain that it can't establish a secure connection.

SSL Pinning with Flutter

Can we really do SSL Pinning with Flutter? Partially. Flutter does not allow for pinning the way we did in Chapter 9 where we calculated the SHA256 hash of our Subject Public Key of the server certificate. Instead, it has the functionality to do SSL Pinning based on CA certificate. Let's see how to implement this. You will need to import the dart.io package by adding this line to your imports:

```
import 'dart:io';
```

Dart provides us with a class called SecurityContext. Now in SecurityContext, you can do two things. First, you can tell any HTTPS connection to not trust the stored root certificates. This means that it will not default to trusting its root CA certificates for each connection. The next thing you have to do is give it some form of a mechanism that it can use to base its decision on whether to connect or not to connect. This is the point where you can include your server's certificate which means it will only trust that. You can find out more about what mechanisms there are to allow your client to trust the server that it speaks with here: `https://api.flutter.dev/flutter/dart-io/SecurityContext/SecurityContext.html`.

To do this, we have to first get a copy of our server certificate. You can do that with OpenSSL, or, if you're using your browser, you can visit the site for which you want to download the certificate, then click the padlock on the browser address bar; then if you're on Chrome or Brave, click the word Certificate (Valid), then drag the icon of the certificate in the resulting dialog window to a directory. See Figure 10-4. This will save the server certificate in DER format. You will need to convert it to PEM format.

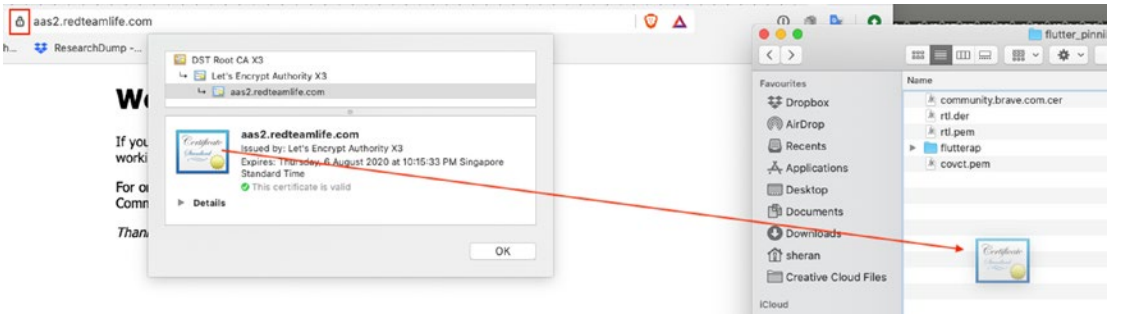


Figure 10-4. *Save the server certificate*

I don't recommend using online tools that you don't trust to convert your certificate. It is always best to handle it yourself. Here is how to convert the certificate using OpenSSL:

```
➔  openssl x509 -in aas2.redteamlife.com.cer -inform der -out rtl.pem
➔  head rtl.pem
-----BEGIN CERTIFICATE-----
MIIFYTCCBEmgAwIBAgISA55ixGZHoNROz2Mrr9bnzH5dMA0GCSqGSIb3DQEBCwUA
MEoxCzAJBgNVBAYTAlVTMRYwFAYDVQQKEw1MZXQncyBFbmNyeXB0MSMwIQYDVQQD
ExpMZXQncyBFbmNyeXB0IEF1dGhvcml0eSBYMzAeFw0yMDA1MDgxNDE1MzNaFw0y
MDA4MDYxNDE1MzNaMB8xHTAbBgNVBAMTFGFhczIucmVkdGVhbWxpZmUuY29tMIIB
IjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAw32mdj0toSw1dwj9cMHSgef0
I4bfGZU0mcYYVhbYzNLHT+6VuZhB3p6O/obplWXsffzvOgcnQYbBZNAUf0CA9Fv6
ickF3LjwURzSMF3Cf1FjX+PQcN37qjSVzwJz1DwkzcW/ECjui6/oBfsBHffIzrPh
G7xij57TrfM/cQfcLP11OBoWTHNArwpKKmk2NU/EFlVdOz4jsGTY/2ywB3/QoKwY
V4uQ2vJw+lxt4vIZosEzhoXOfn2eMwffIClXBHV9F3JsK+OC9lCGA1LujqnP3rBQ
➔
```

Once you have converted the certificate to PEM, create a directory in your Flutter project called assets, as shown in Figure 10-5, and copy your new PEM certificate into that directory.

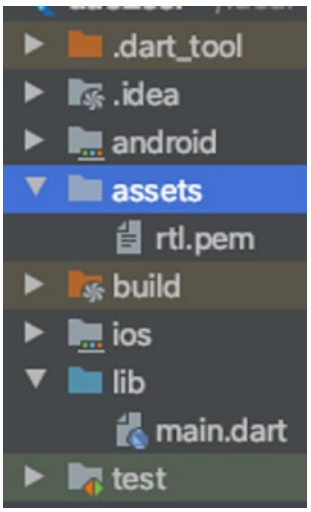


Figure 10-5. *The Flutter project structure with our newly created assets directory*

Next, we will add this new file into our project's assets. Open the pubspec.yaml file in your project and add the following lines under the flutter section:

```
# The following section is specific to Flutter.
flutter:
  assets:
    - rtl.pem

  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true
```

Then modify the code in your _incrementCounter() section to this:

```
01: void _incrementCounter() {
02:     SecurityContext secContext = new SecurityContext(withTrustedRoots:
        false);
03:     var fl = rootBundle.load('assets/rtl.pem').then((value) {
04:       List<int> l = new List();
05:       for(int i = 0; i < value.lengthInBytes; i++){
06:         l.add(value.getInt8(i));
07:       }
08:       secContext.setTrustedCertificatesBytes(l);
09:     });
10:     HttpClient client = new HttpClient(context: secContext);
11:     client.getUrl(Uri.parse("https://aas2.redteamlife.com/")).
        then((request) => request.close()).then((response) =>
        print(response.headers));
12:
13:     setState(() {
14:       _counter++;
15:     });
16:   }
```

Here on line 2, what we're doing is creating a SecurityContext and telling it not to trust or compare against our embedded trusted root CA certificates. Instead, on lines 3 through 7, we load up our file and copy its contents into a new list. Then in line 8, we load that certificate into our SecurityContext.

What we have done so far is to create a SecurityContext that will only allow connections made from our client to talk to a server that presents the certificate we just loaded into memory. Then, we instantiate our HTTPClient and proceed to make a GET request in lines 10–11. By passing our SecurityContext to the client, we tell it that we are only interested in speaking with this one server. Our client will refuse to talk with other servers that do not have this certificate. When you run this code, you should see the HTTP response headers to our site:

```
I/flutter (12143): connection: keep-alive
I/flutter (12143): last-modified: Fri, 08 May 2020 15:06:20 GMT
I/flutter (12143): date: Wed, 03 Jun 2020 12:10:49 GMT
I/flutter (12143): transfer-encoding: chunked
I/flutter (12143): content-encoding: gzip
I/flutter (12143): etag: W/"5eb5756c-264"
I/flutter (12143): content-type: text/html
I/flutter (12143): server: nginx/1.14.2
```

If we changed the host in line 11 to a different one, say `https://google.com`, then we would see an error such as this:

```
E/flutter (20066): [ERROR:flutter/lib/ui/ui_dart_state.cc(157)] Unhandled
Exception: HandshakeException: Handshake error in client (OS Error:
E/flutter (20066): CERTIFICATE_VERIFY_FAILED: unable to get local issuer
certificate(handshake.cc:354))
E/flutter (20066):
```

That's because our client is only interested in speaking with the certificate issued by aas2.redteamlife.com. In this way, you can easily set up SSL Pinning in Flutter. Of course, you can take it a bit further and create a client that only looks at the SHA256 hash of the public key of a certificate, but I leave that up to you to try out.

What we saw here is an attempt to use SSL Pinning on newer and less used software to make it all that much more difficult for a reverse engineer. It is very much akin to security through obscurity because this can be beaten, and I have done exactly that.

Yet again, as with obfuscation, very slight advantage counts, and this extra work that a reverse engineer has to put into bypassing SSL Pinning may be just the deterrent to make them pass over your app.

Golang

Golang or the Go language [https://golang.org/] is an open source programming language that was developed by Google and makes it easy to build performant and portable software. It is a breeze to cross-compile to different platforms and is very fast. I recall how I once rewrote a single API in one afternoon using Go in the startup that I worked for. Our code was a monolithic Java-based file, and we would deploy each large WAR file behind HAProxy load balancers but then only route certain APIs to each WAR file. So we would deploy like ten WAR files, and perhaps two handled one set of APIs, three another set, and the rest handled a third set of APIs. The two that handled one set of APIs would receive location data from thousands of mobile devices every 15 seconds. It would then write this to a database. Because of the load on this API, the VMs running this code had about 90–93% usage, and we were already debugging and profiling on the fly in production but didn't have the time to see it through. That's when I decided to rewrite that component in Go. After I deployed it, each of the servers hosting the Go binary had 2–3% CPU usage. It was completely unbelievable, if I hadn't see it with my own eyes. I loved Go before that, but this one really cemented my relationship with Go. If I could hug a programming language while I slept every night, it would be Go.

The reason I bring up Go in this chapter is because I want to try and implement a native library in Go which I can include inside my Android APK. This library would handle the SSL connection and SSL Pinning functions of my app, and by extension, it would make things a little bit harder for a reverse engineer to decipher. Yes, I know that this is security through obscurity once again, like we did in Flutter, but the more barriers you can put up, the better. Plus, I have been familiarizing myself with the assembly code generated by the Go compiler. The Go compiler used to be written in C, but since 1.5 version of the language, the Go compiler was completely ported over to Go. Keep in mind that Go has a runtime that handles all aspects of program execution, including scheduling, garbage collection, memory management, and so on. This runtime is always included whenever you compile and link your program into a native binary. Therefore, when reverse engineering, you have to keep in mind that the code you're looking at isn't straightforward Go to assembly code. You will also have to navigate through the runtime.

I'd like you to refer back to Figure 10-2 that shows how Flutter apps work with libflutter.so. We are going to replicate this same model, except we will use our own Go library instead of libflutter.so

Gomobile

Go introduced gomobile in version 1.10, and basically what it is is a tool to help you build mobile apps on Android and iOS by writing Go code. Once again, I will not be delving too deeply into Go, except to show you how to get gomobile setup and build a library for Android. To install Go, please follow the installation instructions found on their website [https://golang.org/doc/install].

Once you've got Go up and running on your system, it's time to install gomobile:

```
➔ go get golang.org/x/mobile/cmd/gomobile
➔ gomobile init
```

This should install and initialize gomobile into your environment. Next, we will need the Android NDK because we will have to build a native library. To do this, open Android Studio and go to Tools ➤ SDK Manager ➤ SDK Tools and select NDK (Side by side) and CMake as shown in Figure 10-6.

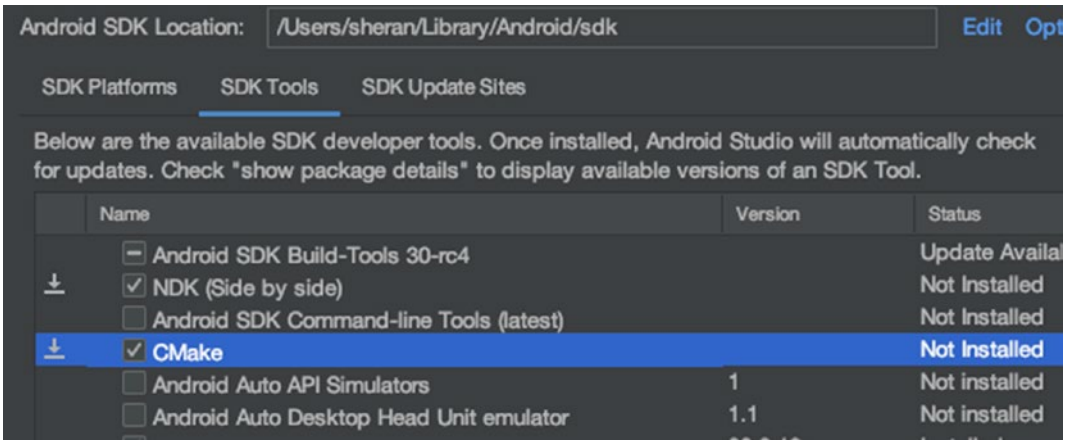


Figure 10-6. *Installing the Android NDK*

When done, click Finish. Now let's write our library in Go. I created a separate directory to hold my Go library. For me, it resides in my Go src folder like so: ~/go/src/github.com/sheran/netutils. I also decided to name my library netutils. You can choose

whatever name you like depending on the use case of course. Now, let's create a Go file and write our code in. I create a file called netutils.go as well and put in this code:

```
01: package netutils
02:
03: import (
04:      "crypto/sha256"
05:      "crypto/tls"
06:      "crypto/x509"
07:      "encoding/hex"
08:      "errors"
09:      "io/ioutil"
10:      "net/http"
11: )
12:
13: func GetVerify(url string) string {
14:      shaPin := "8ccd911bf5ac0ed1bce3d4bb227a2aeed7373b5a7259ec
         162251fb945c9fad57"
15:      config := &tls.Config{
16:           InsecureSkipVerify: true,
17:      }
18:
19:      config.VerifyPeerCertificate = func(certificates [][]byte, _ [
         ][]*x509.Certificate) error {
20:           certs := make([]*x509.Certificate, len(certificates))
21:           for i, asn1Data := range certificates {
22:                cert, err := x509.ParseCertificate(asn1Data)
23:                if err != nil {
24:                     return errors.New("tls: failed to parse certificate
                       from server: " + err.Error())
25:                }
26:                certs[i] = cert
27:           }
28:           cepk, err := x509.MarshalPKIXPublicKey(certs[0].PublicKey)
```

```
29:           if err != nil {
30:                return err
31:           }
32:           pkh := sha256.New()
33:           pkh.Write(cepk)
34:           pubKeyHash := hex.EncodeToString(pkh.Sum(nil))
35:
36:           if pubKeyHash != shaPin {
37:                return errors.New("cannot verify certificate")
38:           }
39:           return nil
40:      }
41:      client := &http.Client{Transport: &http.Transport{TLSClientConfig:
         config}}
42:      req, err := http.NewRequest("GET", url, nil)
43:      if err != nil {
44:           return err.Error()
45:      }
46:      resp, err := client.Do(req)
47:      if err != nil {
48:           return err.Error()
49:      }
50:      body, err := ioutil.ReadAll(resp.Body)
51:      if err != nil {
52:           return err.Error()
53:      }
54:      return string(body)
55: }
56:
```

Caution Be careful that this piece of code does not perform any other certificate verification. You will ideally want to add more verification into this section to ensure that the certificate's roots are also verified.

Let's go over this code briefly. Essentially, this library has one function called GetVerify(). When you write libraries with gomobile bindings, you have to mark your functions as exported or public. In Go, you do this by capitalizing the first letter of your function name. Our GetVerify() function takes one string argument and returns a string – line 13. The argument it accepts is a URL. We are creating a new configuration for our TLS connection called config – line 15. In this, we configure the TLS connection to skip its own verification functions and use our function – line 16. We create this new verification function in line 19. The function is called VerifyPeerCertificate. Within this function, what we do is fetch all the certificates presented to us by the server – lines 20–27. Then we extract the public key from the server certificate, and we calculate its SHA256 hash – lines 28–34. Lastly, we compare it with the SHA256 hash value of our server certificate – lines 36–38. We define our known SHA256 hash in line 14.

Then, we create our HTTP client using the configuration we created – line 41. Then we go on to create our request, send the request, receive the response, and read and return the response body – lines 42–54. Here, for demonstration purposes, we're only working on HTTP GET requests. Also, there's no use of HTTP headers or request body. These can be added as per Go's normal HTTP library found here: https://golang.org/pkg/net/http/. What's important is that you create all your HTTP clients by using the customized configuration and that you initialize the HTTP client using this configuration: `client := &http.Client{Transport: &http.Transport{TLSClientConfig: config}}`

Next, we have to build this library. Using the command line, navigate to the directory where you saved the preceding code. Then, run this command:

```
➔  X gomobile bind -trimpath -o netutils.aar -target=android github.com/sheran/netutils
```

Replace github.com/sheran/netutils with your own path. Essentially, it is the name of your Go package. This should create a netutils.aar package in the current directory. This is your AAR file (Android Archive) to be included in your Android project. Let's analyze this command for a bit. You will notice the -target parameter is set to android. This means you will build the library for Android devices both 32- and 64-bit ARM and x86. By qualifying the target with a / you can further segment the library. For instance, if you only wanted to build an ARM version, you can set your target to android/arm. Or if you only wanted to build an ARM64 version, you can do android/arm64. By doing this, you can obviously shrink your library size, but then you limit it to what types of devices

it can run on. I think the android/arm setting works well for us. Next, it is time for us to build our Android project and use the function we wrote. For this, fire up your Android Studio, and open up the code we last had for the aas2obfuscate app. At the end of it, we add our code that calls our library app like so:

```
01: package com.redteamlife.aas2.aas2obfuscate
02:
03: import android.bluetooth.BluetoothAdapter
04: import android.os.Bundle
05: import android.util.Log
06: import android.widget.TextView
07: import androidx.appcompat.app.AppCompatActivity
08: import com.scottyab.rootbeer.RootBeer
09:
10: import netutils.Netutils
11:
12: class MainActivity : AppCompatActivity() {
13:
14:     override fun onCreate(savedInstanceState: Bundle?) {
15:         super.onCreate(savedInstanceState)
16:         setContentView(R.layout.activity_main)
17:
18:         val testModule = TestModule()
19:
20:
21:         if (testModule.isOddDay()) {
22:             findViewById<TextView>(R.id.oddDay).text = "It's an odd day
                today."
23:         }
24:
25:         val nuts = testModule.getNuts()
26:         val loop : TextView = findViewById(R.id.loop)
27:         for (type in nuts){
28:             loop.append(type+"\n")
29:         }
30:
```

```
31:         val btAdapter = BluetoothAdapter.getDefaultAdapter()
32:         Log.d("aas2obfuscate","BT Adapter address is "+ btAdapter.
            address)
33:
34:         val rootBeer = RootBeer(applicationContext)
35:         if (rootBeer.isRooted()) {
36:             Log.d("aas2obfuscate","Device has been rooted!")
37:         } else {
38:             Log.d("aas2obfuscate", "No root detected")
39:         }
40:
41:         val out = Netutils.getVerify("https://aas2.redteamlife.com")
42:         Log.d("aas2obfuscate",out)
43:
44:     }
45: }
46:
```

Notice the bold lines 10, 41, and 42. This is what you need to do to introduce the code from our Go library into your Android app. But you will also need to make a few more adjustments before you can build the app. First, we have to add the module into our project. To do this, in your Android Studio, go to File ➤ Project Structure. Then, navigate to the Modules section on the left pane of the window that just opened, and add a module by clicking the plus button as shown in Figure 10-7.

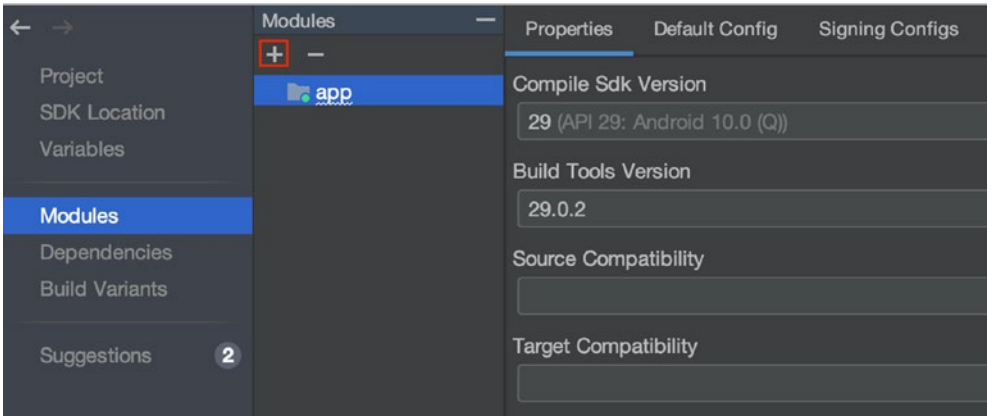


Figure 10-7. *Adding a new module in Project Structure*

You will then be presented with a window containing a list of modules you can add. Find the one that says Import .JAR/.AAR Package, select it, and click Next. Then locate your newly built library. In my case, it is netutils.aar. Put in this location on the path folder as shown in Figure 10-8.

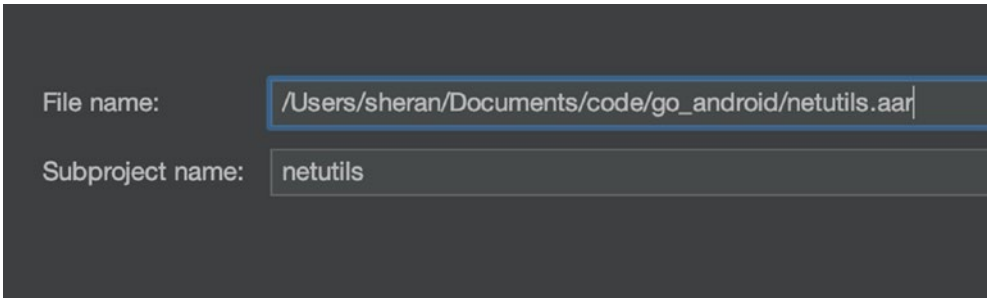


Figure 10-8. *Adding the netutils.aar archive*

When you're done, click Finish. Then back at the Project Structure window, click OK and the archive will be added.

Now we have to add the netutils package to our settings.gradle file. In your project, under the Gradle Scripts, open the settings.gradle file and add this line to the include section:

```
include ':app', ':netutils'
```

Next, we have to edit the build.gradle file belonging to the Module:app. So open that file and add this line under the dependencies section:

```
implementation project(':netutils')
```

For context, I am including my settings.gradle and build.gradle (Module:app) files as follows:

settings.gradle

```
rootProject.name='aas2obfuscate'
include ':app', ':netutils'
```

build.gradle (Module:app)

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"

    defaultConfig {
        applicationId "com.redteamlife.aas2.aas2obfuscate"
        minSdkVersion 26
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            //minifyEnabled true
            //proguardFiles getDefaultProguardFile('proguard-android-
              optimize.txt'), 'proguard-rules.pro'

        }
    }

}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_
    version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.scottyab:rootbeer-lib:0.0.8'
```

```
    implementation project(':netutils')
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Now all that's left to do is build the project and run it on your device and see the results. Here is what happened when I ran the app after redirecting traffic to my proxy (Figure 10-9) vs. when I ran it clean (Figure 10-10).

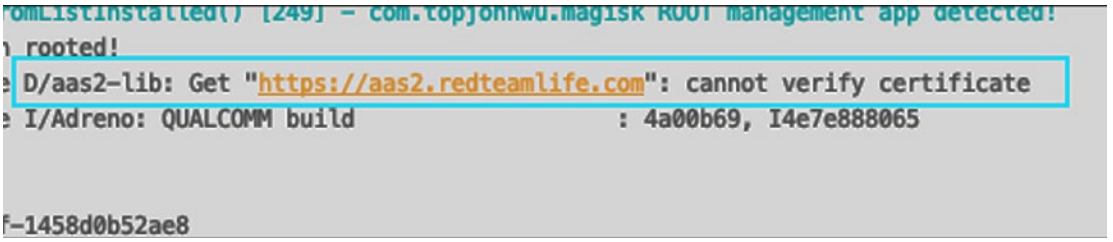


Figure 10-9. *Error in verifying the certificate because we presented the fake SSL cert*

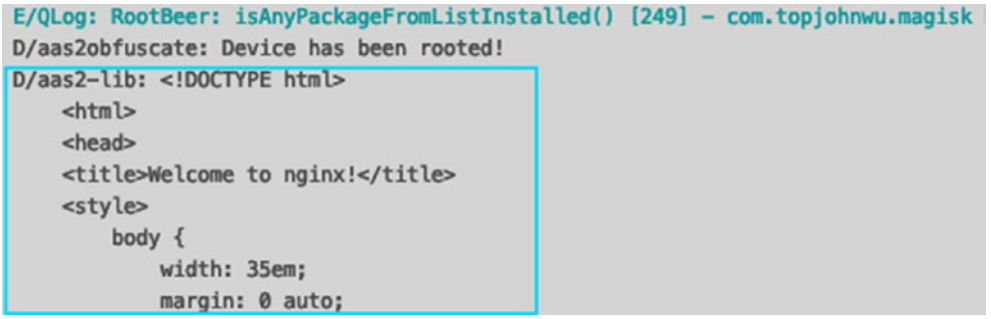


Figure 10-10. *The successfully retrieved HTTP response body from the server*

This demo was a very brief look into how you can consider writing and including your code in your Android apps. Another option, besides C, would be to use Rust [www.rust-lang.org/]. I don't have sufficient research done on using Rust at this moment, but will update whatever research I have on the book's companion website in the future [https://aas2book.com].

Trusted Execution Environment

It would be remiss of me to not include a note about the TEE or Trusted Execution Environment. A TEE is a separate, secure area within the processor of a device. Essentially, it is a hardware-enforced level of isolation for code and data for when sensitive transactions are meant to be executed. Generally, trusted environments are built within the main processors of devices and provide a parallel, secure area referred to as the secure world. The code that runs in this secure world is not visible to nor accessible from the nonsecure world (your Android OS). Figure 10-11 shows the separation between what the ARM processor calls the Rich Execution Environment (REE) or Android and the Trusted Execution Environment (TEE).

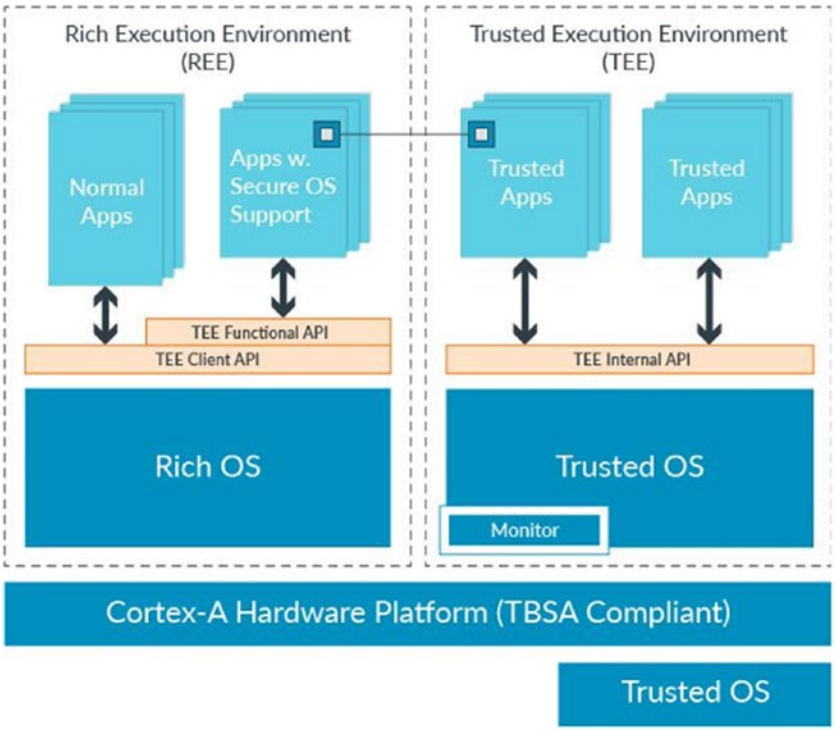


Figure 10-11. *The Arm TrustZone for the ARM Cortex-A processor*

Here's the catch though, in order to leverage the power of this special trusted or secure world, you have to work with a device manufacturer or be one. This sucks because you have a built-in secure area on your device, but you can't actually use it – well, not completely. The Android Keystore which is hardware backed makes use of

the TEE on the device to generate and store secret keys. The trusted application that is running in the secure world in this case is called Keymaster, and it is accessed via a Kernel interface and Abstraction layer. Its architecture is outlined in Figure 10-12. Thus, if you wanted to use the Trusted Execution Environment, one way is by using Google's hardware-backed keystore. Further info can be found here: `https://source.android.com/security/keystore`. If you want to see more about the Android-specific ways to use the hardware-backed keystore, you can find it here: `https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.html`. There's also several examples there.

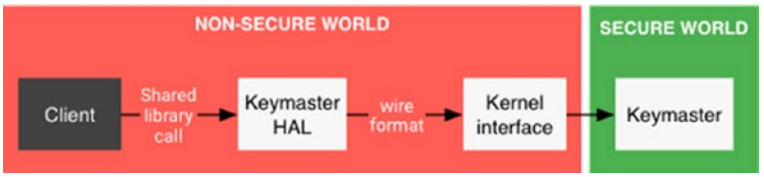


Figure 10-12. *Access to the Keymaster that lives in the secure world*

Another way that you can access the TEE is via a Trusted Application Protection (TAP) SDK from a company called Trustonic. Trustonic has been working in the TEE space for some time now and as a result has had its trusted operating system named Kinibi deployed in over 1 billion devices. The Trusted Application Protection SDK is a development kit that app developers can buy and use in their own apps. The TAP SDK can give app developers a more flexible suite of tools to build functionality that leverages the security of the TEE. One quick way to identify if an app uses the Trustonic TAP SDK would be to look in its packaged native libraries. Figure 10-13 shows an app that uses the SDK. You will notice the three native libraries embedded within the app that facilitates your Android app to be able to implement functionality within the TEE.

Figure 10-13. *An app that uses Trustonic's TAP SDK*

Admittedly, I have not done further research on the Trustonic TAP SDK, nor have I looked too in depth into the TEE. I do, however, am aware that there have been instances where the secure world has been compromised. Any further research that I do will be up on my blog as an extension to this chapter [https://aas2book.com].

Future Evolution of Android

I am not one to speculate on this, nor do I have sufficient evidence that Android will either continue to live or die. I do know for a fact that Google has been working on a new operating system called Fuchsia. The OS is said to be designed from the ground up for security and updatability. The security part of things, I fully understand and appreciate. Since based on Linux, Android has had a fair share of Linux's vulnerabilities transplanted over to it. At Fuchsia's core is the microkernel called Zircon. Zircon contains many of the services, drivers, and libraries that are necessary for it to boot, interface with hardware, load and run user processes, and so on. More information on Fuchsia can be found here: https://fuchsia.dev/.

Fuchsia is an exciting evolution that Android can take. Especially since it is language agnostic, you can write your apps in many languages including Rust and Go. Further, it is built in a manner where you can support new languages so it is possible to support building apps or programs in new languages which is really awesome. I'll be doing more research on Fuchsia as well, and it would pay to definitely keep an eye on its progress.

Principles I (Try to) Live By

I figure rather than preaching to you, I'll tell you some of my paranoia that leads my actions. Similar to how the TEE considers the Android world as insecure, so do I. You may think it is a bit of overkill, but I consider that every part of my app will be dissected and its data inspected and tampered with. The best I can choose to do is to design and develop my app on this principle. Therefore, I sit and think around some core concepts of app design and figure out how to give the attackers the lowest chance of success.

Data

Only store the bare minimum. Let the back end store the rest. Even if you don't have an opportunity for getting accounts brute-forced, build your GET or data retrieving APIs in such a manner as to not leak sensitive personal data.

Encryption on Android, if you are not generating and saving your keys in KeyStore, is most likely going to fail. Whatever you encrypt can be decrypted later if you embed the secret key in your app. You may choose to pull a random string from one of your image assets as a secret key much like how early versions of WhatsApp used to do, but by tracing the encryption functions themselves, it is easy to narrow down where this piece of data is stored. Bottom line is if your app can see it, the attacker can see it.

Network

Even though SSL Pinning can be defeated, use it. Find some new or innovative ways to do pinning, put them in a library, and use them later in your code.

Consider using non-HTTP protocols such as gRPC [https://grpc.io/] as they don't allow a traditional HTTP sniffer or MiTM tool to work. Again, your days may be numbered as there are already tools that do man-in-the-middle attacks on gRPC. There's an upcoming version of HTTP/3 or version 3 of HTTP so it may be conceivable that new security testing or reversing tools will have to be used once again to intercept or tamper with that traffic. As with all cat and mouse games, it's always about trying to stay ahead of the attackers.

Your back ends can be your downfall as well. Get them pen tested regularly or use a bug bounty platform. The usual progression of how an attack takes place is first dissecting the mobile app to determine how it communicates with the back-end server. More often than not, because for some reason developers think their traffic is magically hidden by virtue of its use in a mobile app, there are some abysmal choices made with regard to securing the back end. After the app is generally compromised, the next domino in line is the back-end server. Armed with knowledge of how it talks to the mobile app, it is very trivial to script most of the attacks against the back end. Protect that well.

User Experience

Don't lose sight of what your application is supposed to do. Securing the crap out of a simple app that requires no sensitive data saving nor financial information can be detrimental to its performance and perhaps user experience. Take a look at the big picture first and decide the level of protection you want to give it. Approaching this and most other security-related issues from a user-centric manner will be better.

Wrapping Up

Well, you made it to the end of this book. Thank you. Thank you for taking the time to read it and support it – if you bought it ;) – and thank you for taking an interest in Android Security. There are many, many other Android books that are absolutely brilliant that you should pick up. They are packed full of really good information to allow you to go much deeper into Android. I list two of them here:

- Jonathan Levin's *Android Internals – A Confectioner's Cookbook*, Volume I (I'm eagerly awaiting Volume II)
- Nikolay Elenkov's *Android Security Internals*

I will have sections for each of this book's chapters at `https://aas2book.com`, and I will continue to add to those sections the further research that I do on those specific topics. At that site and the Apress site, you will be able to also send in any errors, omissions, or inconsistencies. Technical books are no longer written to remain static, and the growth of the Internet has allowed a far greater interaction between the author and the reader to a point where technical books are now more fluid. So feel free to engage and let me know where I can improve or point out something inaccurate.

I wish you the very best in your journey toward securing your Android apps. As long as you hold your user's best interests at heart, you should have a good base with which to embark on this perilous journey. Bye for now.

Index

S. Gunasekera, *Android Apps Security*, https://doi.org/10.1007/978-1-4842-1682-8

E

F

G

H

S

T, U

V, W, X, Y, Z