A COMPARISON OF THE USABILITY OF SECURITY MECHANISMS PROVIDED BY IOS AND ANDROID

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

David Maulick

June 2018

COMMITTEE MEMBERSHIP

TITLE:	A Comparison of the Usability of Security Mechanisms Provided by iOS and Android
AUTHOR:	David Maulick
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Zachary Peterson, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Y. Oliver, Ph.D. Professor of Computer Engineering
COMMITTEE MEMBER:	Bridget Benson, Ph.D. Professor of Computer Engineering

ABSTRACT

A Comparison of the Usability of Security Mechanisms Provided by iOS and Android

David Maulick

The Open Web Application Security Project identifies that the number one vulnerability in mobile applications is the misuse of platform-provided security mechanisms. This means that platforms like iOS and Android, which now account for 99.8% of the mobile phone market, are providing mechanisms that are consistently being used in an incorrect manner. This statistic shines a spotlight onto both platforms. Why is it that so many people are misusing platform provided security mechanisms? And is it the platforms fault? The supposition of this paper is that both iOS and Android are not creating usable security mechanisms.

This paper is meant to be a direct response to the number one spot on the OWASP Top Ten Mobile Vulnerabilities list. As a result, our primary goal is to identify whether or not iOS and Android are creating usable security mechanisms. To do this we first proposed an evaluation framework that is tailored to evaluate the usability of mobile device security mechanisms. Then we used it to evaluate seven of the most important and therefore most popular security mechanisms provided by iOS and Android. Through this evaluation we not only hope to develop a clear landscape of overall mobile security mechanism usability, but we also hope to compare the usability across the two platforms.

Overall, it was found that both platforms adequately supported the more popular security mechanisms like key storage and HTTPS. Whereas support for some of the more low-level mechanisms, like encryption and MACs, were often neglected. Such neglect could be seen in a number of different ways; however, the most common neglect came in the form of old documentation, or APIs that are long over do for a rebuild

or increased abstraction. Furthermore, both platforms barely addressed the testing of implementations, despite the fact that testing is arguably the most important part of the software development cycle. Both iOS and Android seldom gave the developer any guidance on verifying the functionality of their implementations.

ACKNOWLEDGMENTS

Thanks to:

- My mother and father for their unconditional support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

Chapter 1

INTRODUCTION

Each year cyber exploits are becoming more consistent, in fact from 2015 to 2017 the number of recorded data breaches have more than doubled [1]. It is common knowledge that hackers have been attempting to exploit desktop operating systems and add-on software for years. Therefore, there has been a lot of effort put into securing such devices. But it is 2018 and 95% of Americans own a cellphone of some kind, and 77% of Americans own smartphones [2]. With this emergence of smartphone popularity comes new and often more severe security threats than ever before. Our smartphones are essentially handheld computers. But unlike our laptops and desktops, we bring our smartphones everywhere we go, and almost never turn them off.

From a networking perspective, these mobile computers are now continuously operating on unknown and often insecure networks that an attacker could be set up on. On top of this, simply bringing our phones with us everywhere we go increases the probability of getting it stolen or losing it. In fact, in 2013 alone it was reported that 4.5 million phones were lost or stolen. In both scenarios it is plain to see that mobile phones are not just the easier target for hackers, but the smarter target. This especially shows true when one considers that mobile devices are being used to handle an individual's most sensitive data. According to the U.S. Federal Reserve, all the way back in 2014, 40% of mobile phone owners with bank accounts were already using mobile banking apps [3]. Using mobile devices as the primary target of an attack is not just the low risk option, it is also showing to be a high reward attack vector.

As shown in the facts of the previous paragraphs, there is an enormous need for continuous improvement in mobile security. OWASP, the Open Web Application

Security Project [4], is one entity that is leading the charge on such defense. OWASP is an open source community that has a very strong reputation for its contribution to the secure software community. In regards to mobile, they are most prominently known for their OWASP Top 10 Mobile Vulnerabilities List. They release this list every two years; this paper will be referring to the 2016 installment of the list as the 2018 list has yet to be released. This list is a fantastic representation of the most common vulnerabilities in mobile security, and should be thought of as a centralized resource intended to give developers and security teams the resources they need to build and maintain secure mobile applications [5].

The number one spot on the OWASP top -10 list is an essential building block of this paper; it is Improper platform usage. Its description identifies that the most common reason for vulnerable mobile apps is that developers are implementing security mechanisms incorrectly. For example, it mentioned that developers often misuse keychain in iOS, or use insecure permissions on Android [4], thereby rendering their app insecure. The placement of this vulnerability as the number one place on the OWASP top 10 list brings the spotlight directly onto the platforms creating these security mechanisms. The main question that comes to mind is: why is it that such a large number of developers consistently implement incorrect solutions for a platform provided mechanisms/API?

The supposition of this paper is that the platforms are not creating security mechanisms that are easy for the developer to use. This means that platforms are not creating security mechanisms that are easy for the developer to use. In fact, judging by OWASPs number one spot, it appears platform provided security mechanisms are quite difficult to implement correctly.

This paper seeks to investigate the previously stated supposition through a comprehensive usability evaluation on platform provided security mechanisms. To do this,

the paper will evaluate security mechanisms associated with three other common vulnerabilities on the OWASP top 10 list: Insecure Data Storage, Insecure Communication, and Insecure Authentication. To do this, the paper will first propose an evaluation framework that will be used to evaluate the usability of a platform provided security mechanism. Then it will use the proposed framework to evaluate the usability of 6 security mechanisms associated with the above identified security vulnerabilities.

This papers evaluation framework is inspired by a framework proposed in a previous iteration of this work. The original framework can be found in Florian Steins paper: A framework for the evaluation of Mobile Platform Support for Implementing Security Mechanisms [6]. While this papers evaluation framework will be different than Steins it will remain comparable. The goal of changing the framework in this paper is to simplify Steins framework into fewer, more simplified fields of evaluation. By doing this we hope the subcategories of the evaluation become more meaningful to the reader. However, we are careful to not change too much about the framework because another goal of this paper is to further legitimize the framework through constant results between this paper and the prior. In order to do this the methodology must remain at its core, similar to the original iteration.

The primary goal of this work is to yield evaluation results that provide insight into how usable the security mechanisms provided by iOS and Android are. The evaluation results are not meant to be a comparison of which platform is better or worse, rather a comparison of how usable they are. Secondly, we hope to contribute to the field of API usability by proposing a robust evaluation framework to assess security mechanisms. Finally, this paper is meant to be an extension of OWASPs mission to enable developers to be more aware and better prepared to create secure mobile applications. Therefore, this paper is meant to add to the open source community, and serve as a useful resource to the average iOS and Android developer seeking to

implement common security mechanisms properly.

In order to focus the scope of this paper, we will be narrowing our efforts on the two largest mobile device platforms: iOS and Android. The mobile device market share has been monopolized by these two platforms [7], therefore it makes sense to limit the evaluation to these two platforms. For iOS we will be evaluating iOS 11; on Android we will be evaluating Oreo. Evaluating iOS 11 makes sense because despite its recent release, 52% of iPhone users have already adopted it [8]. The software generation decision for Android was more convoluted because there is such a wide variation in current Android software. However, Android Oreo was ultimately chosen because it is the most recently deployed and relates to the most up-to-date documentation from Google.

It is important to mention that this paper is not without its weak spots. Likely the biggest issue is that assessing the usability of an API is difficult to quantify and therefore subjective in nature. As a result the ideal way to measure usability would be through survey to get the perspective of many programmers. However, this approach was not realistic for this iteration because of time constraints, and the fact that a survey of this kind would require programmers familiar with iOS and Android. Cal Poly only offers an iOS class currently so finding an adequate survey group was unlikely. Recognizing this flaw in the evaluation is important. However, as stated one of the goals of this paper is to further legitimize Steins evaluation approach. Stein had three people use his framework to evaluate the security mechanisms in his paper. By building on from his framework, another iteration of evaluation that yields similar results will bolster the legitimacy of the framework. Further, discrepancy between the results will highlight potential issues with the framework. Also, in this paper there is one major assumption. We are assuming that iOS and Android have released security mechanisms that are in inherently secure when configured correctly.

In the remainder of this paper we will carry out the research necessary to identify how well iOS and Android support the developer in correct implementation of their provided security mechanisms. First the paper will identify the security mechanisms that will be evaluated in this paper. Then it will outline the framework that will be used to evaluate each security mechanism outlined. Once this is done, the evaluation of each security mechanism will be carried out.

Overall, it was found that both platforms adequately supported the more popular security mechanisms like key storage and HTTPS. Whereas support for some of the more low-level mechanisms, like encryption and MACs, were often neglected. Such neglect could be seen in a number of different ways; however, the most common neglect came in the form of old documentation, or APIs that are long over do for a rebuild or increased abstraction. Furthermore, both platforms barely addressed the testing of implementations, despite the fact that testing is arguably the most important part of the software development cycle. Both iOS and Android seldom gave the developer any guidance on verifying the functionality of their implementations.

Chapter 2

RELATED WORKS

In our background research it was no surprise that (mobile) security in general has been researched quite in depth. However there is only one related work found that shared the same goal as this paper: Florian Steins A Framework for the Evaluation of Mobile Platform Support for Implementing Security Mechanisms [6]. Steins paper primarily seeked to propose an evaluation framework for evaluating the usability of mobile security mechanisms. But in order test out its feasibility, the framework was used to evaluate a number of security mechanisms.

The inspiration for our paper is in direct relation to the OWASP top mobile vulnerability lists number one spot, improper platform usage[9]. It identified the fact that the number one issue in mobile security is the lack of usability in mobile platform APIs[10]. OWASP identified that such APIs include anything from non-security APIs to complex security mechanisms. We used this issue as a traction point that would allow us to identify that usability of security APIs are an issue.

Because the primary focus of this paper is on the usability of security mechanisms. The OWASP top ten list was further used to narrow our scope of which security APIs to evaluate the usability of in this paper. We decided to evaluate the usability of security mechanisms associated with the number two, three and four spot on the OWASP mobile vulnerabilities top ten list.

Because this papers primary goal is to evaluate the usability of mobile security mechanisms, usability evaluation techniques were the primary field of research prior to the start of the experimentation. As Stein stated: being the single author of this work carries with a it a higher likelihood of error, subjectivity and lack of statisti-

cal relevance. Therefore it was deemed worthwhile to research alternate evaluation approaches. Through research there were three usability evaluation approaches identified:

1. Surveys
2. Automated Evaluation
3. Cognitive dimension framework

Each of the above options had its strengths and weaknesses. In Steins paper it was identified that the evaluation of usability by one person is in nature a subjective evaluation. Therefore survey was a usability evaluation technique that emerged as likely the most objective and effective technique. However usability evaluation through survey was quickly debunked because of lack of resources and time constraints. In order to carry out an adequate survey evaluation of usability of mobile security mechanisms it is required to have a participant pool that is familiar with both iOS and Android. This was a major constraint at Cal poly because there is only an iOS class at Cal Poly. Further, actually getting this class to participate would be a challenge in itself that required more preparation.

The second evaluation option that was considered is an automated approach. In other words the evaluation could have been a set of programs that would evaluate the different categories of the framework from a more quantitative perspective. An example of such a program would be counting the number of functions associated with a certain mechanism to indicate difficulty of implementation. This approach would have been ideal from a speed of execution point of view. However this approach was also quickly debunked. Such an approach lacks the descriptiveness necessary for a usability evaluation. Usability in nature is subjective and therefore its evaluation must maintain components of subjectiveness in order to be adequately informative.

As a result a purely quantitative evaluation would be insufficient. A sufficient programmatic approach would require machine learning and/or NLP to account for the many qualitative aspects of usability; this approach however would stand alone as its own thesis.

Like Stein, the final decision was to use a cognitive dimensions framework. The cognitive dimensions framework was first introduced by Thomas Green (University of Leeds) in 1989/1991 and further detailed by Green and Marian Petre a few years later[11]. The paper identified the subjective nature of evaluating usability, so it designed the cognitive dimensions framework to act as a lightweight approach to analyze quality of design or usability.

While this paper ultimately decides to build upon Steins cognitive framework approach for evaluating usability of security mechanisms. Steins framework had a number of pitfalls that this paper seeks to address and improve upon. For example, at times Steins evaluation seemed redundant as it would evaluate subcategories that overlapped. For example, its evaluation of documentation measured the structure and awareness of platform provided documentation. Structure measured how well linked the pages were, while awareness measured how well the pages mentioned other important documents. It was identified that these two fields were greatly similar and could actually be combined into one evaluation. Further there were certain fields that were too difficult to evaluate effectively to bring meaningful results to the study. As a result this paper attempts to maintain a majority of Steins approach, but also attempts to simplify it into a more compact and easy to follow framework. In each effort to simplify, the cognitive dimensions framework[11] was heavily consulted in order to maintain the original foundations proposed in the original proposal of the cognitive dimensions framework. Such decisions are further discussed in the implementation section where the criteria of the evaluation of described in depth.

The overall structure of the usability evaluation was created in relation to the software development life cycle (SDLC)[12]. In order to evaluate the usability of platform provided mechanisms we had to evaluate where in the software development cycle the developer building the mechanism would likely expect help from the platform that built the mechanism. It was identified that there were three main parts where the platform would be able to assist the developer in their creation of the mechanisms. First, when the developer first begins learning about the mechanism, they will require useful and succinct documentation to help them understand the workings of the mechanism. Second, did the platform provide a mechanism that is easy for the developer to use. In other words did the platform create a well designed API. And finally, does the platform help the developer test their implementation. As a result it was identified that our evaluation framework should evaluate the platforms support on how well the documented the API is, how easy it is to implement, and whether it assists in testing the mechanism.

Lastly, one vice of Steins paper was that it often used terms like security mechanism and platform in a vague manner. In this part of the related works the goal is to clear up any obscurity by directly defining how we mean certain terms. Although some of these terms may seem obvious to some readers, each terms use in this paper is vital to the clarity of argumentation. Therefore we are taking this precaution to overwrite any preconceived definitions of the terms.

1. Security vulnerability is a category of security that has been identified as an attack vector for a hacker. Typically will be referring to a vulnerability identified by the OWASP top ten Mobile vulnerability list [9]

2. A Platform is the operating system of a given mobile device. This paper will be evaluating security mechanisms created by Apple and Google. Apples platform is iOS and Googles platform is Android.

3. A Security mechanism is a cover term for any defense against a Security vulnerability. A security mechanism could be any sort of API, feature or control that is used to defend against a given security vulnerability. Note that security mechanism, counter mechanism, counter mechanism API and security API are equivalent in the context of this paper.

Chapter 3

METHODOLOGY

This section is broken up into two main parts. In the first section we will identify the security mechanisms that this paper will be evaluating the usability of. Then in the second section we will propose the evaluation framework being used in this paper.

Table 3.1: Summary of Chosen Security Mechanisms and Vulnerabilities

Idenifier	**Security Mechanism**
Insecure Data Storage	
M-SD-1	Encryption
M-SD-2	MAC
M-SD-3	Key Storage
Insecure Communication	
M-SD-1	HTTPS
M-SD-2	Certificate Pinning
Insecure Authentication	
M-SD-1	Device Credential Authentication

3.1 Introduction to Selected Security Mechanisms

Because the purpose of this paper is so heavily based on the number one vulnerability (improper platform usage) from the OWASP top ten Vulnerabilities list[13], it seemed natural to evaluate security mechanisms that defended against the other major vulnerabilities identified by OWASP. As a result, this paper will be evaluating the usability security mechanisms that are commonly used to counter that of OWASPs second, third, and fourth biggest vulnerabilities in mobile. These vulnerabilities are

insecure data storage, insecure communication, and insecure authentication respectively. In table 3.1, we identify the security mechanisms that we will be evaluating in this paper paired with their respective security vulnerability. The remainder of this section is simply meant to be an introduction to each mechanism covered in this study. If the reader is familiar with these security mechanisms they should feel free to skip this section.

3.1.1 Insecure Data Storage

The number two spot on the OWASP top ten Mobile Vulnerabilities list is insecure data Storage. The section refers to any form of data storage that can be considered insecure, or even unintended data leakage. This paper will assess the usability of three primary security mechanisms that are often used in the secure storage of data: encryption, MACs and key storage.

The first security mechanism evaluated in defense against insecure data storage is encryption. Encryptions purpose is to maintain confidentiality, it does not ensure integrity or authenticity of use. The only purpose of encryption is to scramble the original piece of data or message so that only personnel that have access to the key can access the true meaning of the scrambled data. To maintain confidentiality, encryption encodes a message in such a way that only authorized parties can access it, and those who are not authorized cannot [14].

Without encryption, an adversary would be able to steal all of the information on a users device simply by imaging it on their desktop. Prior to the iOS 8 release, an iPhone was insufficiently encrypted [15][16]. Likewise, until 2014, Android by default stored data on disk in plain text and did not require manufacturers to encrypt data [16]. Back then if an adversary got physical access to any iPhone or android phone, all they had to do was plug it into a computer and they would instantly have access

to all of your personal information.

Nowadays iOS and Android encrypt all of the data associated with their products. However, it is still the responsibility of third party apps to correctly encrypt their user data, and as shown by OWASP, correct encryption is still an issue. Further, An Empirical Study of the Cryptographic Misuse in Android Applications, by Manual Egele and David Brumley, solidifies the lack of effective cryptography use in the Google play store. They identified that 10,327 out of 11,748 in the google play store that use cryptography make at least one error in their implementation [17].

It is also worth mentioning that encryption comes in a number of different forms. Encryption algorithms can use asymmetric keys, where there are two keys but just one is kept private. There are also symmetric algorithms that use 1 single private key. There are also different types of encryption algorithms that can be used as well as varying modes of operation depending on the type of algorithm. Certain configurations have been cracked recently rendering them insecure, yet Android and iOS both still support many insecure algorithms, for example DES. And if one implements one of these insecure algorithms neither platform will stop the developer. These variations will be further discussed in a later section. But ultimately the paper is concerned with the security mechanisms provided by iOS and Android to implement any of these encryption schemes securely.

While encryption is concerned with maintaining the confidentiality of data, MACs are concerned with maintaining the integrity of data and its authentication. As a result, the MAC is our second security mechanism covered in this section. There are a number of MAC algorithms and each are intended to be used as a tool to verify that a given message has not been tampered with by a third party. Some common types of MACs include the NMAC, CMAC, and PMAC. But most popularly used, and likewise the type of MAC typically implemented by iOS and Android, is the

HMAC. As a result the usability of APIs concerning HMACs will be the what we will be focusing on.

An HMAC is essentially a keyed cryptographic hash function. A hash function calculates a fixed-size bit string (a hash, also referred to as digest) from a piece of data often referred to as a message[18]. If the message has been altered in any way while in storage or transit, the hash of this altered message will be different than the original hash; and will thereby identify that the message has been altered[18]. MAC algorithms are typically used instead of cryptographic hash functions because they provide a layer of authentication on top of the integrity verification provided by hash functions.

HMACs authenticate and verify integrity because it makes use of a shared private key. It is used under the assumption that only those with approved access have the key. As a result HMACs are able to not only verify that the data has not changed, it also verifies that the data is coming from the person one expects it to be coming from. If either the private key or the message vary; the MAC will be different and will identify that the data has been tampered with. While a cryptographic hash function does a great job of tracking integrity for an instance like randomized bit flips, MACs are absolutely necessary to track deliberate alteration of data where an attacker may have attempted to hide their changes.

One may notice that encryption and MACs have a common vulnerability between them. Both of their security is only sufficient if their associated private keys are not accessible by the attacker. If the attacker is somehow able to attain the keys they will easily be able to see and alter any data on the device. Therefore, secure key storage is a truly vital part of securely storing any data.

Key storage is a vital component of secure data storage when one realizes how many keys must be protected in order to maintain secure but also efficient experience

on a given mobile device. As a result, Key Storage is the third security mechanism evaluated in this section. A computer does not only need to store keys associated with MACs and encrypted data, computers must also store keys associated with random accounts or passwords. The Android keystore and iOSs keychain are the typical tools used to allow for this extra layer of security. To bolster this both Android and iOS have developed dedicated hardware to work in unison with these keystores to even further bolster the security of the device.

3.1.2 Insecure Communication

The third most common mobile vulnerability is insecure communication. This vulnerability corresponds to any insecurity associated with getting data from point A to point B. The usual risks include, but are not limited to data integrity, data confidentiality and origin integrity [19]. If in transit data can be changed without detection by the receiver this is a major risk. If an attacker can somehow eavesdrop on the communication between point A and B this is another risk associated with the vulnerability. This category also includes offline attacks where an attacker could record communication and then break it later with third party tools.

In this paper we will approach secure communication from the point of view of a client mobile app that is a part of a client-server architecture. This will allow us to more effectively delve into specific portions of the libraries associated with secure communication. Likewise, this is the approach that Stein took, so by maintaining the previous approach our results will be directly comparable.

The secure socket layer (SSL) protocol was originally created by Netscape as SSLv2 in 1994, where it was created to support a secure commerce platform that would operate over the wire[20]. Since then SSL has proven to be a vitally important technique in secure communication. As a result it has since been iteratively fixed up,

patched and re-designed into newer more secure versions, eventually being renamed as Transport layer security (TLS) protocol, hence the confusion between SSL and TLS.

The reason SSL/TLS is so important to this section is that if the entire process of TLS is implemented correctly, from the certificate chain of trust to the public key exchange to the private key encryption; secure communication can be considered highly likely.

Hypertext transfer protocol is the underlying protocol used by the internet to transfer web data across the network. Its use has been the standard for years, however all traffic associated with HTTP is clear text. This leaves all information sent over HTTP wide open for attackers to eavesdrop on. In order to protect against this vulnerability, HTTPS was created to protect web traffic. This was done by implementing SSL/TLS with HTTP, effectively encapsulating the clear text data into an encrypted format. As a result HTTPS is the first security mechanism that is evaluated in this section.

The certificate chain of trust is another vitally important component of SSL/TLS implementation. The certificate chain of trust is essentially what verifies that a given entity one is trying to communicate with is who they say they are. The explanation for how certificates verify connections is one that is overly complicated for explanation in this paper. But the important part to understand is that the integrity of a certificate that is verifying an SSL/TLS connection is absolutely vital. If the certificate is created by an attacker the entire connection will be compromised.

The certificates that ones app will inherently trust are a set of over a hundred certificates located on the OS. The second vulnerability evaluated in this section is Certificate Pinning, and it is implemented in direct response to the lack of trust towards the certificates that are inherently trusted by the OS. Many app developers

have moved to using this mechanism so that their app will only trust one or two certificates that they are in control of. In implementing certificate pinning, the developer can hard code the certificates they trust into their application. This is a somewhat advanced mechanism, but it has proven to be very effective.

3.1.3 Insecure Authentication

The only security mechanism being evaluated in this section is the use of local device credential authentication. OWASP identifies that many apps carry issues concerning authentication of their users. This includes cases like failing to identify the user at all, failure to maintain user identify of the user, and plain incorrect user authentication. As a result, we chose to evaluate passcode and biometric verification. These are two mechanisms primarily supported by the platform. But it is suggested that developers building third party apps also make use of such authentication.

3.2 The Evaluation Framework

The evaluation framework that we are proposing is meant to evaluate how well iOS and Android support the average developer in implementing security mechanisms. As a result it was designed in a manner that would reflect the typical software development cycle. In this way its evaluation is broken up into three main parts and these parts are outlined in the remainder of this section. The first part will evaluate the documentation that the platform provides. The second section will evaluate how usable the actual mechanism is. And the final section will assess how well the platform supports the developer in testing the mechanism.

3.2.1 Documentation

When developers wish to leverage a new feature supported by the native platform, they will often look to the documentation as a starting point. The official documentation is typically where one will go to achieve this step. Therefore, this first section of the framework evaluates how useful the security mechanism documentation provided by the two major platforms, iOS and Android, are. Because this work aims to measure iOS and Android support, the only documentation being considered is that which is provided by the platform. No third party information is considered; e.g.. stack overflow, etc. This is because the use of third party support sites often promote incorrect direction and insecure code as result of their lack of expert review[21].

Both iOS and Android have two, of what we refer to as, document databases that we will consider valid support pages. First, both iOS and Android have a more modern developers guide[22][23] that we refer to as the platforms primary database. The documents in the developer guides are typically up-to-date and almost always contain actual direction in implementing code. As a result, documents from the primary database are almost always prefered in this study. Both also have what we refer to as a secondary database. This database contains numerous original guides and documentation that iOS and Android were originally built on. For iOS, we refer to this as their legacy database[24]; it is the primary resource for platform information that typically explains the underlying functionality of Mac OS and iOS. As a result most of the information is old, and written in C and Objective-C. Androids secondary database is Java docs. Android is built from Java and as a result a lot of its low-level libraries, like the cryptography libraries still refer to the original Java documentation.

In order to assess the documentation provided by a platform, we will take a generalized approach to compare each platforms documentation. This generalized approach will evaluate each platforms documentation on three categories of evaluation: cover-

age, structure, and solution. The details of each of these subcategories are outlined in the remainder of this section.

The first subcategory, coverage, evaluates each platforms documentation on whether or not all of the necessary information for a given security mechanism is available to the programmer. In this subcategory we identify a generalized set of documents that helps to serve two purposes.

1. If a security mechanism has a document that resembles each document in the identified set of documents we are looking for, it achieves documentation coverage.

2. Also, this set of documents allows for a generalized way to refer to the documentation throughout the remainder of the paper.

There are a total of six documents in the document set. The first two documents in the set are constant for all security mechanisms evaluated in the paper: the programming home page[22][23] and the security home page[25][26]. Both Android and iOS have one, and they serve as great starting points for any knowledge search that needs to be done when creating a secure Android or iOS app.

The remainder of the documents in the set were identified through an iterative evaluation process of the security mechanisms in this paper. It was found that when a security mechanism was perfectly documented, the platform provided a document that represented each of the following pages existed.

1. A vulnerability homepage associated with the security mechanism

2. A security mechanism home page

3. A pictured solution for implementing the security mechanism

4. Complete Library documentation for the solution

The vulnerability homepage(1) explains what the vulnerability is, and the risks associated with it; it will typically allude to the associated counter mechanisms. The security mechanism home page(2) discusses how the mechanism works and explains why a developer might want to implement it. The solution(3) is a page with a pictured implementation of the security mechanism code. Please note that there is an entire other subcategory that will evaluate the quality of the provided solution. And the final document in the set is the library documentation(4) associated with the identified solution. This last document is actually a set of library documents provided by the platform. For this subset of documents to be deemed existent, all API/functions from the solution document must be in the library docs provided by the platform. Also note that in some cases, certain documents in the set may correspond to the same link. For example, the solution page could be the same page as the mechanism landing page.

The scoring for the coverage of a given security mechanism starts at a full score of three out of three points. If any of the functionality in the solution is not documented in the library, the coverage score will receive an automatic zero out of three. Otherwise, the total score for the coverage category will start at a max of 3, and will be docked one point for every other document in the set that is missing. Therefore the coverage score can receive a score of 0, 1, 2, or 3. The number value awarded to the coverage score is directly associated with how many of the first 3 documents in our set exist. Please note that this tally excludes the security and programming home pages because we already know these exist, as well as the library entries.

In the actual evaluation, a table like the one in table 3.3 will often be referenced. The check mark represents that the given document from the document set is provided by the platform. If the check mark is missing, it means that the platform failed

to provide a page that adequately represented that given document. In the below example it is shown that all documents were provided in the coverage evaluation with the exception of the iOS security mechanism homepage.

Table 3.2: Coverage Example

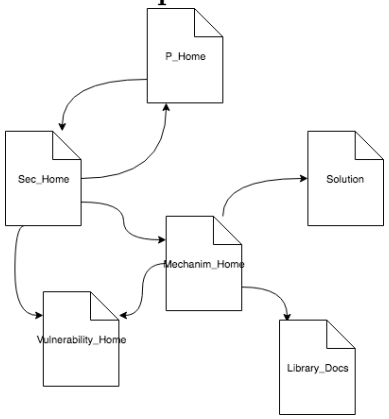
	iOS	Android
Vulnerability home page	✓	✓
Mechanism home page		✓
Solution	✓	✓
Library docs	✓	✓

In building the document set for the coverage evaluation it was often found that a majority of the documents in our document set existed. But saying a document exists is much different than saying a document exists and is discoverable. Often in our evaluation it felt as though certain pages were not easily found, in fact, sometimes it felt as though they were only found because their existence was important to the research being done in this paper. It felt as though if we were developers in the same situation, as opposed to researchers, we wouldve likely given up the search for such information much earlier.

As a result the second category of the documentation evaluation is structure. This category evaluates the document set identified by the coverage section for how discoverable each of the associated documents are. This is done by evaluating how closely interlinked each of the documents in the document set are. Since the web of connections is difficult to describe we will use a graph to represent interconnection of the document set like the one below. Each document will represent a node in the graph, and a direct link between the pages will be represented by an edge in the graph. The programming home page will be represented by the P_Home node

and the Security homepage will be represented by the Sec_Home node. Vulnerability homepage, security mechanism homepage, and solution will be represented by the Vulnerability_Home, Mechanism_Home, and Solution nodes respectively. And finally, the Library docs will be represented by the Library_Docs node.

Figure 3.1: Example of Structure Graph



This evaluation is scored starting at a max score of 2 and is docked for each violation of the following guidelines: if all documents in the document set can be linked together into a web of direct connections the structure evaluation will receive a full score. If just one document in the document set is two edges away, or in other words, indirectly linked from a document in the remainder of the document set, structure will receive a half score of 1. If a single document is two or more edges away from all other documents in the document set, structure will receive a score of zero. Likewise, if more than one document is more than one edge away from the majority of the document set, structure will receive a zero. Please also note the special case where the document set is split into two subsets of three documents each. If this occurs, the security mechanism will receive a half score for structure. Likewise, if the only connection between the two subsets is through the library docs document set, the security mechanism will again receive a half score on structure. This is because a path through the subset of library documents does not imply an intuitive direct link.

The final subcategory of documentation is the quality of the solution. As stated previously, a solution is one of the 6 documents needed in the document set. Therefore its existence and discoverability is already tested. This section is therefore concerned with the overall quality of the solution document. This section serves as an explicit focus on the solution document. This was deemed appropriate because it was found that when documentation provides a clear, complete, secure, up-to-date, intuitive solution of exactly how to implement an API, success rates of implementations sky-rocket [27]. The solution category is scored on a scale of 0 to 3 based on these three qualitative characteristics:

1. Pictured, secure code on a single page
2. Complete and clear instruction
3. The solution is up-to-date

The solution evaluation for each security mechanism will start at a max of 3. Then for each of the above require characteristic that it does not meet, the score will be decremented by 1 point. However, if the code provided uses deprecated libraries it will receive an automatic score of zero. This is because it is terrible practice to implement code that is not being tested continuously[28].

Note that the actual details of the solutions implementation is not of concern in solution evaluation. This measure is only concerned with evaluating that the respective platform has provided a legitimate solution. Chapter 3.2.2 will evaluate the actual implementation of the solution in depth.

Comparison with Steins Documentation evaluation

There are a number of differences between this works documentation evaluation and Steins approach. Likely the most apparent difference is its simplification. Steins

approach consisted of five sub-evaluations, whereas our approach only has three. This was achieved by removing Steins consistency evaluation and incorporating components of the awareness evaluation into our coverage and structure evaluation.

Stein argued that it was necessary to measure how consistent document instruction was because when there are inconsistencies in documentation the material is exponentially more difficult to understand when compared to an overall consistent document set[6]. It is important to recognize that our work does not disagree with such an idea. However, through analysis of Steins results it became clear that there was no efficient and scientifically sound way to measure overall consistency of a large set of documents by hand. Because of this the results for the consistency section felt unnecessary and lacking in meaning to the overall documentation assessment.

In Steins documentation assessment, coverage, structure and awareness were also subcategories evaluated. The coverage section sought to check that there was one page associated with the introduction of the security mechanism, and one page to document the APIs for the security mechanism. The structure evaluation sought to evaluate how closely linked the documents associated with coverage were. Finally, awareness sought to identify whether common documentation, like that of the programming homepages and security homepage, created awareness for the need of security mechanisms by directly talking about or linking to them.

Upon evaluation of these three categories it became clear that the evaluation of the documentation often felt loose. The above three categories were very closely related to one another and having them separated made the logic path to their numerical scores uncertain in the eyes of the reader. In this paper, we attempt to tighten up that logic path by encompassing the effect Stein was going for with his awareness category into our structure evaluation category. By expanding the document set identified by the coverage evaluation and then directly evaluating linkage between the documents

in that set, the same takeaways from the original awareness category are addressed.

3.2.2 Implementation

As stated in the documentation section, an up-to-date, quality solution is often one of the most valuable pieces of documentation a programmer can come by. However, even with a solution, one cannot just copy and paste code into a project and expect the respective security mechanism to work. The programmer must work to personalize the solution so that it works with their specific project. This section of the framework attempts to evaluate the implementation stage of the software development process. To do this, the section will assess exactly how easy the APIs and controls associated with a given security mechanism are to use and implement into a project.

The first criterion of the implementation section measures the level of abstraction associated with a given security mechanisms implementation. When an API is adequately abstracted it is a joy to use. The functionality of the API will feel intuitively related to the reason that the developer decided to read about it in the first place. And its functionality will hide the highly technical details that may cause confusion, while highlighting the details that the developer is looking for. By measuring the abstraction of a given API it is easy to tell whether the API has been designed with the user of the API in mind. Therefore reflecting a major component of an APIs usability.

We recognize that in some cases an API can be overly abstracted to a point that the API is not flexible enough for the user. However, such a case was never an issue in the APIs that this paper dealt with. Therefore, the more abstracted an API for a given security mechanism is, the higher score they will receive for these criteria. A security mechanism will receive a perfect score on abstraction if the mechanism is abstracted as much as reasonably possible.

In order to evaluate abstraction, we will consider two factors associated with the solution found in the documentation assessment. First, how many implementation choices does the developer need to make that are directly associated with configuring the security mechanism API. A choice in this context is something like choosing a parameter for a method associated with the security mechanism API. For example, in implementing encryption, which encryption algorithm you use is a choice that is directly associated with the mechanism, and therefore is a choice that will be considered in the abstraction criteria. The second factor has to do with how difficult each choice is. This difficulty is measured by evaluating how closely related the choice is to the high level intent of the security mechanism. With respect to the encryption example, the high level intent of the mechanism is to encrypt data. If such implementation simply requires the use of a single method of this form: encryptData(dataToEncrypt) -¿ encryptedData, the implementation can be considered fully abstracted. A non-abstracted encryption implementation may require the developer to implement everything down to the encryption algorithm[6]; thereby receiving an abstraction score of zero. The scoring for the abstraction criteria will be scored out of a max score of three. It will receive a score of zero if it is insufficiently abstracted, one if moderately abstracted, and two if highly abstracted.

The above abstraction assessment adequately evaluates how much effort a developer must exert in order to configure a given security mechanism API to their liking. However, it does not evaluate the extra effort one must exert in order to incorporate the mechanism into their project. We define this extra work done outside of the security mechanism configuration as overhead work. And we will use the minimality criterion to evaluate how much overhead work is required for a given security mechanism implementation. Such a category is vital because it was found that implementations that require a lot of overhead work often scare off developers[27]. To better understand this category, consider the following example:

A developer wants to implement password authentication into their app. The actual code directly associated with the password authentication security mechanism is easy to configure and can be written in 2 or 3 lines of code. However, password authentication requires a UI for the user to enter their passwords into, and there is no user interface provided with the API. As a result, in order for the developer to implement password authentication they must also must write the code for a keypad UI from scratch.

Such an implementation would receive a very good score in terms of its abstraction level since the configuration of the security mechanism itself was so simple. However, the need to build an entire UI to type the passcode into identifies that the mechanism has a lot of overhead work. As a result, such an implementation would receive a zero in its minimality score. Like abstraction, the minimality criteria will also be scored out of three points. It will receive a score of zero if overhead work is greater than or equal to that of the work associated with the security mechanism configuration work evaluated by the abstraction criteria. It will receive a full score of 2 if the overhead is reasonably small. And a one if the amount of overhead work is determined to be anywhere in between the work associated with high and low scores of the minimality criteria.

The abstraction and minimality criteria adequately evaluate APIs for the amount of work that must be put into implementing a security mechanism correctly. However, in the words of Joshua Bloch, a good API must also be hard to misuse[27]. What this means is, a good API should be designed in a way that handles incorrect implementations elegantly and safely. And if a developer does missimplement the API, it ideally should be obvious to the developer. And above all, if the mechanism is implemented incorrectly, its mis-implementation should never undermine the underlying security that the app already employs.

To measure this property of a security mechanism API we have created the robustness criteria. To evaluate the robustness of a security mechanism we will consider the results of two error in implementation of each security mechanism. We recognize that the identification of these two potential errors impose a dimension of bias into the evaluation, however the only way to get around this is by evaluating all possible errors. This however is out of the scope of this paper.

Table 3.3: Robustness Matrix

	Mechanism Security	System Security
Certain	0/3	0/3
Plausible	1/3	0/3
Unlikely	2/3	1/3
Rare	3/3	2/3

In order to evaluate the robustness of each error, the risk matrix above will be utilized. In evaluating each error, we will first identify which part of the application security could be affected by the error: the mechanism security or the underlying security. If the error can affect both the security mechanism security and the underlying security, we will evaluate the table as if the error will affect the underlying security. Next, we must identify how likely the error is. Once these two characteristic of the error are identified the score for robustness can be calculated. This is done by identifying the cross section cell between the likelihood of the errors occurrence and which part of the application security the error could effect. Please note that although this criteria evaluates multiple errors, the final score for robustness is the lowest robustness score of the chosen error evaluations. Also worth noting is that the max score for a potential error that could affect the underlying security of the app is a 2/3, this is because any risk to underlying app security is a major issue.

Comparison with Steins Implementation evaluation

Unlike the documentation section, all three of the criteria used in this papers implementation evaluation are directly related to criteria identified in Steins paper. It was felt that Steins Abstraction, minimality, and robustness criteria actually did a fantastic job of evaluating the overall usability of a security mechanisms implementation.

The one change made in this section was the removal of a criteria called consistency. This criterion was extremely similar to the consistency criterion removed in the documentation section. He identified that inconsistencies often create confusion in implementation. This paper agrees with such a statement. However, in evaluating Steins results we found that these criteria, like the consistency criteria in the documentation section, yielded results that did not add much insight into the overall evaluation. Plus, the methodology in evaluating consistency felt inadequate and therefore the results didnt feel reliable enough to report on.

Also worth noting is that we made some slight changes to the robustness table. The table as a whole maintains the same scoring pattern, it is just smaller.

3.2.3 Testing

As previously mentioned, the evaluation framework in this paper is meant to mirror that of a typical software development cycle. After a developer has learned about the security mechanism API and implemented it, the software should be tested to verify its functionality. One could argue that the testing phase is solely the developers responsibility, not the platforms. However, the idea behind this evaluation is to identify which platform, iOS or Android, best supports the developer in implementing the security mechanism. As a result, if the respective platform does help the developer test their implementation of each security mechanism, the robustness

of the implementation can be guaranteed with more certainty. When a developer can guarantee the robustness of an implementation, the respective implementation is, by default, more usable.

The first and only criteria of the evaluation is the addressed criterion. The purpose of this category is to simply evaluate whether the respective platform supplies the developer with the appropriate tools to test the functionality of the given security mechanism. Testing tools can vary greatly depending on the security mechanism. Whatever the means of testing is, the evaluation will be scored out of 2. If the platform addresses testing of the security mechanism in any of its documentation, the criteria receives one point. Then if it is provided, it must be verified that the testing material sufficiently tests the implementation. If it does the criteria receives a full score of 2/2. If it is not at all addressed in the documentation set the criteria will receive a 0/2.

Chapter 4

EVALUATION

4.1 Insecure Data Storage

The iOS and Android provided security mechanisms associated with OWASPs 2nd most common mobile vulnerability, insecure data storage, will be evaluated. The security mechanisms being evaluates in this section are encryption, MACs and key storage.

4.1.1 Encryption

Documentation

Both iOS and Android receive full marks on their coverage for encryption as all of the necessary document types for both platforms are provided.

Table 4.1: iOS and Android Encryption Coverage

	iOS	Android
Secure Data Storage home page	✓	✓
Encryption home page	✓	✓
Encryption Solution	✓	✓
Library docs	✓	✓

Recall that documents for iOS and Android may be admitted into the document set as long as they are in their respective platforms primary or secondary database. Because of this, the coverage evaluation yielded interesting results for encryption. Compared to all other coverage evaluations, the encryption coverage evaluation found

the highest number of documents, for both iOS and Android, originating from their respective secondary database. This is interesting because it shows that neither platform has made much of an effort to build upon their original encryption documentation. All iOS documents with the exception of the library entries are from its legacy database. And Android is just one off from iOS, having both its encryption landing page and solution residing in Javadocs. For reference, no other coverage evaluation in this paper had more than one document from their set residing in the respective platforms secondary database.

Figure 4.1: iOS Encryption Documentation Structure

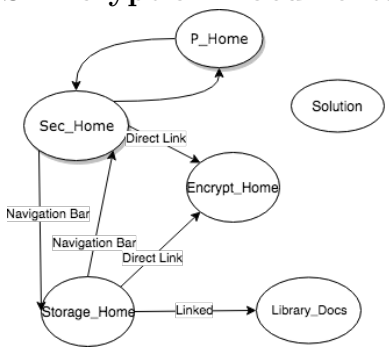
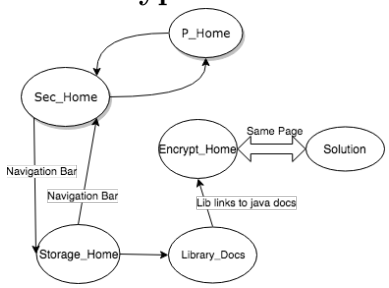


Figure 4.2: Android Encryption Documentation Structure



The layout of the iOS encryption document set was well connected with the exception of the solution. The solution turned out to be a stand-alone Objective-C project called CryptoCompatability[29]. It was mentioned as a useful existing project on the encryption homepage, however no functioning link accompanied the mention. As a result it was only found through a Google search. Despite its isolation the project

was quite useful. CryptoCompatability[29] was intended to teach developers about the low-level C cryptography library commonCrypto. As a result, it not only ran examples for symmetric and asymmetric encryption, but also for MACs and numerous other cryptographic operations. It even supplied the developer with test data to verify correct implementation of cryptographic mechanisms.

Since the iOS solution was more than one link away from the remainder of the document set, the security mechanism would typically receive a score of zero on its structure. But this evaluation turned out to be a special case. The solution title was mentioned in the security mechanism homepage as a very useful project for learning about iOS encryption. It just wasnt linked. As a result the iOS encryption structure evaluation will receive one out of two points.

Interestingly for Android, they have left pretty much all of their useful cryptography documentation to be found in Javadocs, their secondary database. As a result, there is a very clear split between their introductory document set pages and their encryption specific documents. The most convenient way to find their cryptography documentation is by going through the library documentation subset. As a result of this split, Android will receive half points for its structure score.

The iOS solution is found as a standalone Xcode project. The project is written up-to-date objective-C code, and is secure. The instructions associated with the solution are simply comments within the project. As a result, the iOS encryption solution will receive a full score on its solution. However, it is worth noting that since the directions are commented into the code, the code is slightly less readable. The solution would likely be a lot easier to follow if there were a separate document with instructions and code snippets to accompany the project.

The Android encryption solution is conveniently on the same page as the Android encryption home page. This page is titled the Java Cryptography Architecture (JCA)

and contains information regarding all cryptographic libraries supported by Java. Android simply adopted Javas cryptography API for all of its cryptographic use cases. At the top of this page is a table of contents that makes it simple to jump directly to the content that you desire. For example, encryption is clearly linked at the top of it. The only issue with this layout is how it provides code solutions. Because the page is so large, the encryption solution and its instructions are fragmented. Although the solution is still secure, up to data and well annotated; it feels as though it is too spread out. Since this issue is similar to the requirement that states the solution must be presented on one page, the Android encryption solution is docked one point. As a result, it receives a score of two out of three on its solution.

Encryption Implementation

As stated in the documentation evaluation, cryptographic documentation for both iOS and Android have been somewhat neglected. This is because neither platform has spent much time building upon their original implementations. Apple requires the developer to work with a low-level C library called commonCrypto[30] to handle a majority of cryptography needs. And Android still requires their developers to work with the original Java Cryptography Architecture (JCA) [31] that they first adopted in their original creation of the platform.

The first step in both asymmetric and symmetric encryption for Android is getting an instance of the KeyGenerator[32] and the Cipher[14] for the specific type of encryption. Both steps require using the respective classs content provider. To get an instance of a given class from their content provider the developer must use the respective classs .getInstance() method. When retrieving an instance for the KeyGenerator, one must be careful to pass the correct algorithm. Likewise, the getInstance request for the cipher requires a parameter that is referred to as the transformation. The transformation takes on the form algorithm/mode/padding. As long as the algorithm

being passed to the key generator and the cipher content providers are compatible, the encryption code below will work.

Listing 4.1: Symmetric Encryption (Android - Java)

```
1 KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
2 SecretKey secretKey = keyGenerator.generateKey();
3 Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
4 cipher.init(Cipher.ENCRYPT_MODE, secretKey);
5 GCMParameterSpec gcmParameterSpec = new GCMParameterSpec(0, cipher.
     getIV());
6 byte[] dataToEncrypt = ...
7 byte[] encryptedData = cipher.doFinal(dataToEncrypt);
```

Listing 4.2: Asymmetric Encryption (Android - Java)

```
1 KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA");
2 KeyPair keyPair = keyGenerator.generateKeyPair();
3 Cipher cipher = Cipher.getInstance("RSA/NONE/OAEPPadding");
4 cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPrivate());
5 byte[] dataToEncrypt = ...
6 byte[] encryptedData = cipher.doFinal(dataToEncrypt);
```

Once an instance of the cipher and the key are created all one must do is initialize the cipher with the specified key, and its mode: init(int opmode, Key key). Then using the doFinal method the cipher will encrypt the input data returning the encrypted data. Also notice that decrypting is quite simple too. The exact same process is used for decryption except the developer must change the opmode in the cipher initialization to DECRYPT_MODE.

As identified, there are many configurations that must be decided upon when encrypting data. And as OWASP identifies[33], there are a number of configurations that are inherently bad implementations of encryption. Some common errors include

picking an insecure algorithm, picking a key length that is too small or even using a non cryptographically secure PRNG for the key. Because of the numerous possibilities that can arise in encryption implementation, abstraction of the respective API is likely the most important factor in assessing an encryption APIs usability.

Because there are so many parameters that an encryption API could leave up to the developer an ideal implementation in the scope of this paper would be one that requires no input from the developer. The encryption API call would be secure by default. However, Android chose not to do this, likely because they wanted to leave space for flexible implementation for the many experienced programmer that use the API. Instead Android chose to create a thin layer of abstraction that only requires the developer to specify the algorithm, mode of operation and padding. Upon further investigation this degree of abstraction actually makes a lot of sense. It still allows experienced programmers to implement flexible encryption designs. But at the same time, considers less experienced programmers, and only reveals the need for decision on three of the more common characteristic of encryption. Still though, Android receives a half score on its abstraction. The Android encryption implementation for symmetric and asymmetric encryption is quite simple to incorporate into an existing project. As a result, it will receive full marks on minimality.

Although a number of the parameters for Android encryption are abstracted, the API still requires the developer to know about a number of complex characteristics of encryption. The Android developer must specify the algorithm, mode and padding for the cipher, as well as the algorithm for the key. Despite the fact that Android Studio will warn the developer in certain cases: like when ECB mode is used, typically it will not. As a result, any error in parameter specification will render the entire encryption security mechanism useless. Such an issue is likely since Android still supports a number of insecure encryption algorithms[34]. Therefore Android receives zero points for Encryption implementation robustness.

In contrast to Android, and somewhat frustratingly, the iOS implementation for encryption varies across symmetric and asymmetric encryption. Symmetric encryption is implemented through the commonCrypto[30] API; a low-level C library. And asymmetric encryption is implemented through the Certificates, Keys, and Trust services API.

Listing 4.3: Symmetric Encryption (iOS - Objective-C)

```
1  uint8_t * symmetricKey = NULL;
2  NSData * symmetricKeyRef = NULL;
3  symmetricKey = malloc( kCCKeySizeAES128 * sizeof(uint8_t) );
4  memset((void *)symmetricKey, 0x0, kCCKeySizeAES128);
5  int result = SecRandomCopyBytes(kSecRandomDefault, kCCKeySizeAES128,
       symmetricKey);
6  symmetricKeyRef = [[NSData alloc] initWithBytes:(const void *)
       symmetricKey length:kCCKeySizeAES128];
7  uint8_t iv_tmp[kCCBlockSizeAES128];
8  memset((void *) iv_tmp, 0x0, (size_t) sizeof(iv_tmp));
9  result = SecRandomCopyBytes(kSecRandomDefault, kCCBlockSizeAES128,
       iv_tmp);
10 NSData* iv = [NSData dataWithBytes:(const void *)iv_tmp length:
       kCCBlockSizeAES128];
11 size_t outLength;
12 NSData * dataToEncrypt = ...
13 NSMutableData *cipherData = [NSMutableData dataWithLength:
       dataToEncrypt.length + kCCBlockSizeAES128];
14 CCCryptorStatus result = CCCrypt(
15         kCCEncrypt,                    // operation
16         kCCAlgorithmAES128,            // algorithm
17         kCCOptionPKCS7Padding,         // options
18         symmetricKeyRef.bytes,         // key
19         symmetricKeyRef.length,        // keylength
20         iv.bytes,                                    // iv
```

```
21          dataToEncrypt.bytes,            // dataIn
22          dataToEncrypt.length,           // dataInLength,
23          cipherData.mutableBytes,        // dataOut
24          cipherData.length,              // dataOutAvailable
25          &outLength);                    // dataOutMoved
```

Listing 4.4: Asymmetric Encryption (iOS - Objective-C)

```
1  NSMutableDictionary *keyPairAttributes = [[NSMutableDictionary alloc
       ] init];
2  [keyPairAttr setObject:(__bridge id)kSecAttrKeyTypeRSA forKey:(
       __bridge id)kSecAttrKeyType];
3  [keyPairAttr setObject:[NSNumber numberWithInt:2048] forKey:(
       __bridge id)kSecAttrKeySizeInBits];
4  SecKeyRef publicKey = NULL;
5  SecKeyRef privateKey = NULL;
6  OSStatus status = SecKeyGeneratePair((__bridge CFDictionaryRef)
       keyPairAttr, &publicKey, &privateKey);
7  size_t cipherDataSize;
8  uint8_t *cipherData;
9  const uint8_t dataToEncrypt[] = ...
10 size_t dataLength = sizeof(dataToEncrypt)/sizeof(dataToEncrypt[0]);
11 encryptedDataSize = SecKeyGetBlockSize(publicKey);
12 encryptedData = malloc(cipherBufferSize);
13 OSStatus status = SecKeyEncrypt(
14          publicKey,          // key
15          kSecPaddingOAEP,    // padding
16          dataToEncrypt,      // dataIn
17          (size_t) dataLength,// dataInLength
18          cipherData,         // dataOut
19          &cipherDataSize);   // dataOutAvailable
20 return cipherData;
```

Unlike the Android API for encryption, iOS actually does provide a default for the encryption mode of operation: CBC and GCM. One other bright spot is that the system does use a cryptographically secure PRNG. However, compared to Android, the iOS encryption implementations, especially the symmetric implementation are barely abstracted. The symmetric implementation requires the developer to specify every other parameter of encryption aside from the mode of operation. Despite asymmetric implementation being quite a bit simpler to implement our score must reflect the lowest of the two. As a result, the iOS encryption abstraction score will receive a zero because of the difficulty of implementation for symmetric encryption.

The overhead for asymmetric encryption using the Certificates, Keys, and Trust services API is quite minimal. All of the steps are well documented in the documentation and can be easily implemented by a swift developer because it is written in objective-C. However, in order for a developer to implement symmetric encryption they must be familiar with C memory management, pointers and other C programming patterns. Because symmetric encryption requires the developer to be familiar with C, the overhead work is considered to be quite high, but still slightly less than that of the decisions that must be made in configuring the security mechanism. As a result of this, iOS encryption will receive a score of one out of two on its minimality.

The robustness for iOS encryption will receive a zero as a result of its symmetric encryption implementation. The primary issue with the symmetric encryption API is that it requires the developer to write C code. It is simply unrealistic to expect swift developers to be comfortable implementing in C. As a result of the necessity for C code, data vulnerabilities like buffer overflows are now an issue. Such an issue runs the risk of undermining underlying system security. Because it is plausible for the underlying app security to be affected, the robustness of iOS encryption receives a zero. Also worth noting is that nearly every property of the encryption configuration is left up to the discretion of the developer. Like the Android assessment Xcode will

not warn the developer of incorrect encryption specification and this further decreases the robustness of an iOS encryption implementation.

Encryption Testing

Encryption is a difficult security mechanism to test because it is near impossible to identify that the transformed data has been scrambled in a cryptographically secure manner. In fact, the only effective way for a developer to verify that their encryption implementation has been implemented correctly is by verifying with a known test input and output. Therefore, in order for the respective platforms to fully satisfy the encryption addressed criteria, the platform must supply the developer with test input and output data for the developer to use in unit tests of their implementations.

Unfortunately, Android does not support the developer at all in testing the encryption implementation. They do not supply the developer with any test data. Apple on the other hand does support the developer in testing the cryptographic implementations through their cryptographic project CryptoCompatability. The project is an OS X project that not only contains a decently large set of cryptographic methods that can be used in iOS project. It also has complimentary input and output test data. Therefore iOS receives a full score on its addressed score and Android receives a zero.

Just to further test the reliability of the iOS test resources, and verify that the Android implementation was correctly we tested our encryption implementations with NISTs Cryptographic Algorithm Validation Program (CAVP). Luckily both the Android and iOS implementations were correct.

4.1.2 MAC

Documentation

The coverage evaluation for iOS receives a score of two out of three because a page that could represent the landing page for message authentication codes (MACs) did not exist. Android on the other hand fully satisfied the coverage requirements.

Table 4.2: iOS and Android MAC Coverage

	iOS	Android
Secure Data Storage home page	✓	✓
MAC home page		✓
MAC Solution	✓	✓
Library docs	✓	✓

Since MACs are another cryptographic feature it turned out that their solution was actually located in the same project as the encryption solution. However, in contrast to the encryption evaluation, it was not mentioned anywhere that the MAC code could be found in the project labelled cryptocampatability. In fact the only reason this solution was found was because we came across it in our encryption evaluation. In the case of encryption, a developer would have at least had a chance at finding this code since it was mentioned on the encryption home page. However, MACs for iOS have no such home page, and likewise have no mention of the project. As a result, the MAC security mechanism will receive a score of zero on its structure for iOS.

Figure 4.3: iOS MAC Documentation Structure

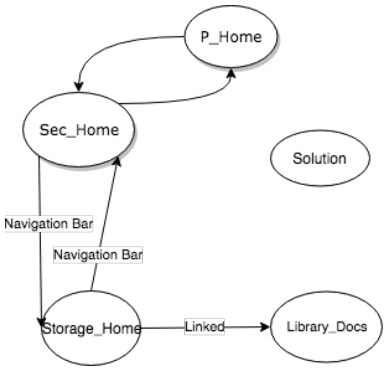
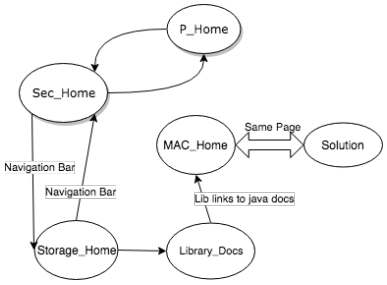


Figure 4.4: Android MAC Documentation Structure



Interestingly the Android graph for MAC structure exactly resembles the Android graph for encryption structure. This is because Android uses the Javadocs cryptography architecture documentation to present all things cryptography in the Android library. As a result both encryption and MAC are clearly presented in the same Javadocs documentation with clearly defined landing pages and solutions. Unfortunately, like encryption, the java doc cryptography architecture docs are only linked to the other documents in the set through the library subset. As a result, Android will receive 1 out of 2 points for its structure.

The solution provided by iOS for MACs thoroughly resembled that of the solutions we identified in the encryption section since this solution was part of the same project called Cryptocompatability. Like those encryption solutions, the iOS MAC solution was pictured, secure and up-to-date. The only downside of the MAC solution was that the discussion was through comments in the project. As mentioned in the encryption

section we would have preferred to have a page where code and instruction were separated. However, a project with instructions written as comments inline with the code satisfies the requirements for complete instruction. As a result iOS receives a full score on solution.

The Android solution is found on the Java cryptography architecture guide(JCA)[31]. As stated in the encryption section, the document maintains a table of contents at the top of it. So despite how large it is, it is easy to navigate to a given topic. For example, it is easy to navigate to the section on MACs. However, the actual code for MACs is scattered across the page, and despite it all being on one page, it feels as though the solution is split across multiple pages. As a result the Android solution for MACs will receive a score of two out of three since the solution feels fragmented. Aside from this fact the code provided is secure, up-to-date, and has clear associated instruction.

Implementation

Like Android encryption, MACs are supported by the Java Cryptography architecture (JCA), implemented as javax.crypto. As a result, the MAC implementation follows the almost exact same implementation pattern as that of Android encryption. The developer must first use a content provider to get an instance of a keyGenerator and an instance of a Mac object. The Mac and keyGenerator request only requires the message authentication code algorithm as an argument. While the concept of a keyGenerator is simple, one may want to think of the Mac object as being similar to the cipher object described in the encryption section, but it will calculate a MAC for the inputted data instead of encrypted data. It is also important to note that the algorithm specified for the keyGenerator must match the algorithm specified for the Mac object like it had to for encryption. Once the key and Mac objects are created the mac is initialized with the secret key and the developer can use the dofinal()

function to calculate the MAC for the inputted data.

Listing 4.5: MAC implementation (Android - Java)

```
1 KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA256");
2 SecretKey secretKey = keyGenerator.generateKey();
3 Mac mac = Mac.getInstance("HmacSHA256");
4 mac.init(secretKey);
5 byte[] hash = mac.doFinal("String␣to␣Hash".getBytes());
```

Despite the fact that the MAC implementation for Android is implemented using the same design pattern as encryption, it will be rewarded a higher abstraction score. This is because in Android encryption there were three characteristics that a developer that is unfamiliar with encryption would have had to research: algorithm, mode of operation and padding schemes. As a result of that evaluation, Android encryption was awarded a half score on abstraction. In the case of MACs all that the developer must specify is the MAC algorithm. Since there is only one decision that must be made in this implementation and any others are abstracted away, the Android abstraction score for MACs will receive a full score.

Like for encryption, Android studio will seldom notify the developer that they are making use of an insecure hash algorithm when implementing an HMAC. For example, Android studio does not warn against using MD5 in MAC implementation. Since Android still supports insecure algorithms, such a error is considered plausible. And since there is only one argument that needs to be decided on, as opposed to the three required by Android encryption, MAC for Android will receive one out of three points for robustness.

Like symmetric encryption, MACs on iOS are supported by the commonCrypto library. As a result the developer must do a fair amount of up front work in C. Then once all of the initialization is done the developer must call the commonCrypto

function CCHMAC() which will calculate and place the MAC into the functions last argument.

Listing 4.6: MAC implementation (iOS - Objective-C)

```
1 uint8_t * tmpKey = malloc(CC_SHA256_BLOCK_BYTES * sizeof(uint8_t) );
2 memset((void *)tmpKey, 0x0, CC_SHA256_BLOCK_BYTES);
3 int result = SecRandomCopyBytes(kSecRandomDefault,
     CC_SHA256_BLOCK_BYTES, tmpKey);
4 NSData * keyAsData = [[NSData alloc] initWithBytes:(const void *)
     tmpKey length:CC_SHA256_BLOCK_BYTES];
5 NSString* stringToHash = @"StringToHash";
6 NSData *dataToHash = [stringToHash dataUsingEncoding:
     NSUTF8StringEncoding];
7 NSMutableData *hash = [NSMutableData dataWithLength:
     CC_SHA256_DIGEST_LENGTH];
8 CCHmac( kCCHmacAlgSHA1, [keyAsData bytes], [keyAsData length], [
     dataToHash bytes], [dataToHash length], [hash mutableBytes]);
```

Despite the difficulty of preparation to implement MACs in iOS, the level of abstraction is not awful. Generating a Mac for a given piece of data only requires one API call to commonCrypto. And this call only requires one decision: which MAC algorithm to use. Since this is the same case as Androids MAC abstraction, iOS will receive a full score for its abstraction.

The iOS minimality score for the MAC implementation is a whole different story compared to its abstraction score. Similar to the encryption implementation, the MAC implementation requires the developer to be familiar with C design patterns like memory management. As a result there is a quite a bit of set up in C in order to generate a MAC for a given segment of data. Unlike the encryption implementation however, the amount of overhead work heavily outweighs the amount of work considered in the abstraction evaluation. As a result iOS MAC minimality score will

receive a zero out of two.

The configuration for iOS MACs is quite comparable to that of the Android MAC implementation. It requires the developer to specify the algorithm for the MAC implementation, while not enforcing the use of secure algorithms in their respective environments. If this were the only identifiable error, iOS would receive a half score for MAC robustness. However, since there is the added likelihood that the C overhead work could introduce an underlying system insecurity; iOS will receive a zero for its MAC robustness score.

Testing

The testing evaluation for MACs is very similar to that of the encryption evaluation. Android provided no support whatsoever for testing MACs. Therefore Android received a score of zero on its testing being address. Apple does actually provide testing data for the implementation of SHA-1 in its cryptocompatability project like it did for encryption. However, it unfortunately does not provide test data for SHA-256. Because of this iOS will receive a half score on its testing being addressed. Like with encryption we verified the MAC generation with test data from CAVP[35] and found that both Android and iOS implemented MAC generation using SHA-1 and SHA-256 correctly.

4.1.3 Key Storage

Documentation

Unlike the documentation for encryption and MACs, Key storage for both iOS and Android is quite thorough. The document mapping shows that both iOS and Android will receive a full score for their coverage of the topic.

Table 4.3: iOS and Android Key Storage Coverage

	iOS	Android
Secure Data Storage home page	✓	✓
Key Storage home page	✓	✓
Key Storage Solution	✓	✓
Library docs	✓	✓

One thing to note in regards to the iOS documentation is that there were duplicates for each of the security mechanism specific documents in the document set. As discussed in the methodology, section 3, iOSs documents are valid candidates for the document set if they come from one of the platforms two document databases: the developer guide or the legacy database. Oddly for the case of iOS key storage, there were documents from both the developer guide and the legacy database that could justifiably represent every single document in the document set. Although this mostly bolstered the certainty of full coverage of Key Storage related content, such duplication often made our selection for the most reliable information difficult. With that said, all necessary information was easily found in the developer guide. So such difficulty in choosing the correct documents likely would not have been an issue for the average developer. The average developer would likely have found everything they need in the developer guide, and not even realize the duplicate information.

Figure 4.5: iOS Key Storage Documentation Structure

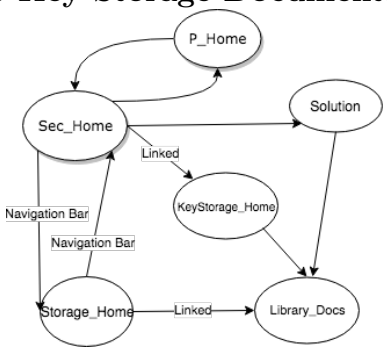
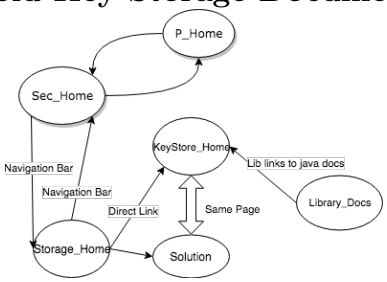


Figure 4.6: Android Key Storage Documentation Structure



The iOS key Storage structure will receive a full score for its structure. This is a result of the fact that the security home page was well linked to the Key storage documentation. The secure data storage homepage, the key storage homepage and the solution are all only indirectly linked across the library docs. But because the security homepage directly linked to each of these pages the iOS structure still receives full points.

The odd iOS structure is a result of two issues in regards to the iOS documentation. First of all the secure data storage home page document for iOS is located in the legacy database while the remainder of the documents in the document set are located in the developer guide. The Legacy documents almost never link directly to the developer guide since they are typically created before the developer guide. The secondary reason is that Apple associates introductory material on the keychain with storing user information. As a result, the only solutions linked to the introductory keychain material is how to store a password in the keychain. Its introductory material does not discuss its use case for a cryptographic key. In order for a developer to find the solution for using the keychain with cryptographic keys, the developer must look at the certificates, key, and trust services API documentation directly linked at the security home page.

The Android Key Storage Structure will also receive a full score on its structure. All pages are appropriately represented and linked through the Android developer

guide. Android also conveniently has a number of solutions pasted onto the KeyStore home page making an implementation especially simple. Also worth noting, Android has directly linked extra examples that show how to use the Keystore with different types of keys at the KeyGenParameterSpec library entry. Such examples include RSA key pairs, AES keys, and HMAC keys. The KeyStore homepage also links to an Android Studio project with an example of how to use the keyStore.

Android provides numerous usable and secure examples of how to use the Keystore. On the Key storage homepage they show how to create cryptographic keys and how to put them into the key store. Plus this page also links to a number of variation solution examples on the KeyGenParameterSpec library entry. It also links to an example project and more examples on the actual KeyStore API documentation. One thing to note is that many of the examples automatically add the key to the keystore upon its creation. However, only the example in the KeyStore API documentation does a simple store and fetch of an existing key. Regardless each of the examples are very useful. The solution being considered in our evaluation is the one that stores an existing key. This example is located on the keystore home page and is secure, up-to-date and has clear instructions. Therefore Android Key Storage receives a full score on solution.

Like the Android documentation, iOS actually shows multiple implementations of how to use the keystore as well. It gives an example of how to store and receive the key associated with a password as well as how to store a cryptographic key. It also has a sample project in the legacy documentation, not linked to the document set however. The solution associated with the storage of a cryptographic key is the solution being examined in this section. It is secure, up-to-date and has clear instructions. It even links to how to create cryptographically secure keys. As a result iOS will also receive a full score on its solution.

Implementation

Because both iOS and Android provided so many usable solutions we decided that we would assess the simplest case. In this section we evaluate the case when a developer is attempting to store an already existing key into the platforms respective key storage mechanism.

Key Storage is vital to a secure but enjoyable user experience because it securely stores sensitive key data, while maintaining quick access to such keys in the right situations. For example, a user would not want to enter their password for a given app every time they open it. However, they do not want the system to store such sensitive data unless it is secure. The respective platforms key storage system is the entity that makes sure that such a key is securely stored and only reachable by the appropriate app. With this architecture, when a key is added to the keyStore it is automatically encrypted and stored. After this, the only time that the key may be accessed again is when the phone is unlocked by the device password; and then only the specific app that has access to it may be granted access to said key.

The Android implementation is quite simple. All the developer must do in terms of configuration for use is call the KeyStore content provider for an instance of the Android KeyStore. As a result the only decision of the developer is to specify which keyStore they would like to use by passing the name of the requested keystore. In order to use the Android key store the developer must pass AndroidKeyStore as a parameter to the getInstance() method of the KeyStore. Once the instance is created one can easily add and query the Android KeyStore with the setEntry and getEntry methods. One can simply add the key to the Keystore by passing a chosen alias name and the key as a parameter to the setEntry() method. Then to retrieve the key, simply load the KeyStore and use the getEntry() method with the alias as a parameter to retrieve that same key.

Listing 4.7: Key Storage implementation (Android - Java)

```
1 // store key
2 SecretKey secretKey = ...;
3 KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
4 KeyStore.SecretKeyEntry secretKeyEntry = new KeyStore.SecretKeyEntry
     (secretKey);
5 keyStore.setEntry("NameForKey", secretKeyEntry, null);
6
7 // retrieve key
8 keyStore.load(null);
9 SecretKey secretKey = (SecretKey) keyStore.getKey("keyName", null);
```

One could argue that Android could further abstract the content provider so that by default it would return the Android KeyStore. However I do not think such increased abstraction is necessary. The content provider is written this way so that it is simple to read. If the keyStore specification were omitted, it would be less clear where exactly the keys are being stored, and it would also cover up the fact that one can make use of multiple Keystores. As a result we will grade Android Key storage to be fully abstracted. There is also virtually no overhead for this implementation. And finally the implementation will receive a full score on its robustness as there is no place where one would expect the developer to make a mistake that could undermine security. Even if they chose a different keystore the key would remain secure.

The process to store an iOS cryptographic key into keychain is also quite simple. It will be considered completely abstracted because the only decisions made by the developer are the choice of key and the keys associated type. The programmer must specify these characteristic by creating a dictionary with keys kSecClass and kSecValueData. Once the dictionary is created there is one API call necessary to add the key to the keychain. By passing specified dictionary to the method SecItemAdd the key is safely added to the keychain. To retrieve the data the exact same process is

carried out, but with the SecItemCopyMatching() keychain API call. Like the Android API call, pretty much all of the information about how the key is being stored is abstracted away. This creates an easy configuration for the developer, but maintains security of the keys because the keychain will only return to the apps with the correct permissions.

Listing 4.8: Key Storage implementation (iOS - Objective-C)

```
1  // store key
2  NSData *key = ... ;
3  NSMutableDictionary *attributeDictionary = [[NSMutableDictionary
       alloc] init];
4
5  [attributeDictionary setObject:(__bridge id)kSecClassGenericPassword
       forKey:(__bridge id)kSecClass];
6  [attributeDictionary setObject:key forKey:(__bridge id)kSecValueData
       ];
7
8  OSStatus status = SecItemAdd((CFDictionaryRef)attributeDictionary,
       NULL);
9
10 // retrieve key
11 NSMutableDictionary *queryDictionary = [[NSMutableDictionary alloc]
       init];
12
13 [queryDictionary setObject:(__bridge id)kSecClassGenericPassword
       forKey:(__bridge id)kSecClass];
14 [queryDictionary setObject:(_bridge id)kCFBooleanTrue forKey:(
       __bridge id)kSecReturnData];
15
16 CFTypeRef result = NULL;
17 OSStatus status = SecItemCopyMatching((__bridge CFDictionaryRef)
       queryDictionary, &result);
```

While the actual implementation does not require any C programming, the keychain API is written in C. As a result the developer must carry out a fair amount of overhead work to properly prepare the dictionary and result data to be compatible with the API call. For example, there is a quite a bit of type casting required in the lead up to the call. It is however noted that none of this overhead requires advanced experience in C. Because of this, key storage for iOS will just be docked one point on its minimality.

Despite the overhead C preparation the iOS implementation will still receive a full score on its robustness. Like for Android, there is no error that would render the security mechanism or that of the underlying system security insecure. The only likely errors identified was incorrect dictionary format and/or incorrect type casting, which would both throw exceptions upon the API call.

Testing

Both iOS and Android will receive not applicable (N/A) for their testing being addressed for key storage. This is because the testing of key storage is somewhat implied with its correct implementation. In order to test it we simply stored a key in the respective platforms keystore and attempted to retrieve the key to verify its existence in the Keychain. This was confirmed to work in both Android and iOS. In order to further test the keychain we tested to make sure that keys were only accessible by the correct apps. Through this effort it was confirmed that the keystore operated as expected, only granting access to the apps that store the specified keys for both iOS and Android. Steins paper took this testing phase one step further to investigate the effects of a jailbroken or rooted device on the functionality of the keystore. This was not performed in this iteration of the evaluation.

4.2 Insecure Communication

In this section we will assess the usability of security mechanism APIs that address OWASPs 3rd most common mobile insecurity; insecure communication. This section will evaluate the the platform provided implementations for HTTPS and certificate pinning.

4.2.1 HTTPS

Documentation

As the below document mapping indicates, both iOS and Android have great coverage on the information necessary for implementing their provided HTTPS mechanism. This was somewhat expected as HTTPS is essentially a giv in in the current technological climate. Still however, it is great to prove that both platforms are both thoroughly documenting a somewhat trivial implementation. As a result both will receive a full score on their HTTPS coverage.

Table 4.4: iOS and Android HTTPS Coverage

	iOS	Android
Secure Comms home page	✓	✓
HTTPS home page	✓	✓
HTTPS Solution	✓	✓
Library docs	✓	✓

As seen below, the structure of the Android documentation set is extremely effective. The security homepage has a guide and samples section that directly links to the secure communication page which is located in a spot that is easy for developers to find. Once on the Android secure communication home page one can directly link

to the HTTPS mechanism page and the solution through the navigation bar as well as through a link in the secure communication text. Also the Android Solution and mechanism page are one in the same. As a result, Android will receive a full score for its HTTPS documentation structure.

Figure 4.7: iOS HTTPS Documentation Structure

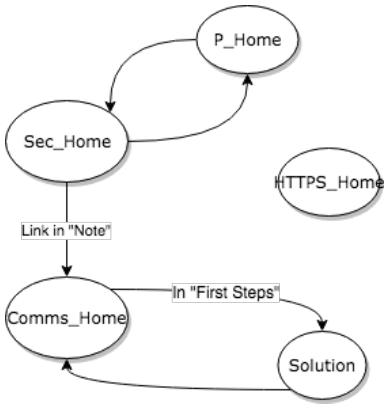
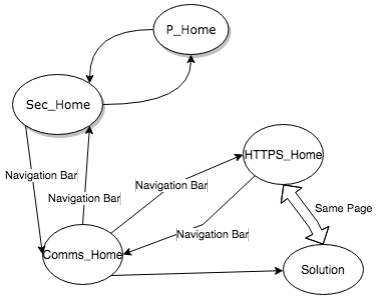


Figure 4.8: Android HTTPS Documentation Structure



The structure for iOS was good with the exception of the location for the HTTPS homepage. Because all pages except for the HTTPS homepage were directly linked, iOS will receive half marks. The documentation for iOS HTTPS again identifies a frustrating feature of the iOS documentation. Apple has two two primary locations for their iOS documentation. As discussed, there is a legacy database that contains a lot of great information of the fundamentals of the iOS platform. However it is often outdated implementing deprecated APIs. And there is the more modern developers guide that seems to have continuous maintenance. In this evaluation it was seen that

all but the HTTPS homepage were linked together in the more modern developers guide. This is not of huge concern because one is able to implement HTTPS without the HTTPS home page; however if the developer is unfamiliar with HTTPS concepts, this indirect linkage is a major drawback to understanding the importance of using HTTPS.

The solution for Android HTTPS receives a perfect score. The solution is conveniently located in Androids HTTPS homepage, it is pictured, secure, and up-to-date. It also has clear and complete instructions inline with the solution to increase clarity.

The iOS HTTPS solution receives a score of 3 out of 3 also. The solution provided by iOS is pictured, secure, up-to-date and provides detailed instruction with its code sample. However it is worth noting that the iOS solution instructions are a bit harder to follow than Androids because there are two ways to implement HTTPS in iOS. One can implement a simplified use of iOSs HTTPS API which makes use of the singleton shared instance for the URLSession class. This is what the first half of this document discusses. However any experienced programmer will be quick to recognize that implementing this singleton class results in restricted flexibility in regards to their networking settings. For example, it would not be possible to implement certificate pinning in the correct manner with the singleton instance. Apples documentation does handle this issue with grace by clearly identifying such drawbacks with an explanation and code snippets of the more flexible implementation. But as a result the documentation requires more reading than likely necessary; especially when compared to Androids documentation.

Implementation

As stated previously, there were two ways to implement iOS. In both cases iOS requires the developer to use what they refer to as the URL loading system, which is there framework that interfaces with standard internet protocols like HTTPS.

The solution provided by iOS can be found on a page titled Fetching Website Data into Memory. As stated prior we are using the more flexible version of the HTTPS implementation. The simplified version uses a singleton class for quick use, while the more flexible version requires the developer to set up a delegate for the URLSession instance being used. The following code is pictured on the Fetching Website Data into Memory page that was identified as the iOS HTTPS solution page.

Listing 4.9: Creating a URLSession that uses a Delegate (iOS - Swift)

```
1 private lazy var session: URLSession = {
2     let configuration = URLSessionConfiguration.default
3     configuration.waitsForConnectivity = true
4     return URLSession(configuration: configuration, delegate: self,
          delegateQueue: nil)
5 }()
```

Listing 4.10: Using a Delegate with a URL session data task (iOS - Swift)

```
1  var receivedData: Data?
2
3  func startLoad_delegate() {
4          loadButton.isEnabled = false
5          print("In startload 1")
6          let url = URL(string: "http://www.facebook.com/")!
7          receivedData = Data()
8          let task = session.dataTask(with: url)
9          print("In startload 2")
10         task.resume()
11         print("In startload 3")
12     }
13
14 // delegate methods
15
16 func urlSession(_ session: URLSession, dataTask: URLSessionDataTask,
```

```
    didReceive response: URLResponse,
17                  completionHandler: @escaping (URLSession.
                        ResponseDisposition) -> Void) {
18      guard let response = response as? HTTPURLResponse,
19          (200...299).contains(response.statusCode),
20          let mimeType = response.mimeType,
21          mimeType == "text/html" else {
22              completionHandler(.cancel)
23              print("Cancelled.\n")
24              return
25      }
26      print("Success.\n")
27      completionHandler(.allow)
28  }
29
30  func urlSession(_ session: URLSession, dataTask: URLSessionDataTask,
        didReceive data: Data) {
31      self.receivedData?.append(data)
32  }
33
34  func urlSession(_ session: URLSession, task: URLSessionTask,
       didCompleteWithError error: Error?) {
35      DispatchQueue.main.async {
36          self.loadButton.isEnabled = true
37          if let error = error {
38              // handleClientError(error)
39              print("ERROR: Delegate URL session completed with error
                    !!: \(error)")
40          } else if let receivedData = self.receivedData,
41              let string = String(data: receivedData, encoding: .utf8)
                    {
42              self.webView.loadHTMLString(string, baseURL: task.
```

```
                currentRequest?.url)
43          }
44      }
45  }
```

The above code interacts with a UI that consists of a button and a webview. The idea behind the app is that when the button is pressed the webview loads www.example.com using HTTPS. Note that this is how iOS implements HTTP also. However as a result of iOSs app transport security (ATS) update in the release of the iOS 9 SDK, if the developer tries to access example.com using http, an exception will be thrown and the app will not run.

Figure 4.9 above shows how the developer should create a URLSession object so that it uses a delegate instead of the shared singleton URLSession class. The second snippet of code immediately follows the first on the solution document and shows the delegate methods that must be implemented in order to make use of a URLSession delegate.

On first glance, the sheer amount of code needed to implement HTTPS for iOS was surprising. However the solution page that we identified comes with not only a complete and secure implementation but it is extremely well explained. Moreover, the code supplied as the solution is exactly the code necessary to implement HTTPS. The only decision that a developer would need to make in configuring this API is which website they would like to interact with. As a result of this simplicity of implementation the iOS HTTPS security mechanism is identified to be highly abstracted, and will receive full marks for this category.

Androids implementation for HTTPS is quite simple, and Android illustrates its simplicity through the below 4 line code snippet located in their HTTPS landing page:

As can be seen below, since Android does not use a singleton class or a delegate to implement HTTPS its implementation is intuitive even to a programmer is not familiar with such creational design patterns. HTTPS for Android will receive a full score for abstraction because like iOS the only decision that a developer must make in implementing the security mechanism is which website to connect to.

Listing 4.11: Opening a URL connection (Java)

```
1 URL url = new URL("https://wikipedia.org");
2 URLConnection urlConnection = url.openConnection();
3 InputStream in = urlConnection.getInputStream();
4 copyInputStreamToOutputStream(in, System.out);
```

Both iOS and Android will also receive full marks on minimality for their respective HTTPS mechanisms. Although on first glance one may argue that iOS has an oddly large amount of code associated with its implementation, all of that code is directly associated with the mechanism configuration. Therefore it is out of the scope of a minimality evaluation and associated with the abstraction evaluation. Plus, delegation is a common design pattern in iOS development. To a beginner it can seem as though this large amount of code for such a simple task adds complexity. But in reality it allows for very effective fine grained configuration of a given HTTPS connection.

Interestingly the robustness of the two mechanisms vary. As identified in the abstraction evaluation, there is only one choice that must be made for both the iOS and Android implementation. In each the developer must specify the desired URL. Mistyping the URL to be written with http:// is a potential error that this paper deems quite plausible. Such a error would render the mechanism useless, but the underlying systems security will remain in tact. As a result Android receives 1/3 for robustness as this is a realistic outcome for the Android API. Apple on the other had

is able to score full marks on HTTPS robustness because of the recent release of App Transport Security (ATS). This capability was released with the iOS 9 SDK, and can be thought of more of a review policy for iOS app code as opposed to an update. If it recognizes that your app is using HTTP instead of HTTPS it will literally throw an exception at runtime and stop the app from running. As a result the iOS HTTPS implementation can be considered extremely robust and will receive full marks.

Testing

Both iOSs Xcode and Androids Android Studio provide the developer with quite robust and useful test suites. Apples Xcode includes a tool called Instruments,[36] which apple describes as a flexible performance and analysis tool. This tool can be used not only to test things like memory and energy usage, but also network statistics and performance. As a result it can be used somewhat effectively to track whether HTTPS is being used by showing whether or not the traffic is coming from port 443. Although ports can be mapped anywhere, this is a decently effective way of identifying that the connection is using HTTPS. Since it does not fully assess the connection though, the iOS addressed score for HTTPS will receive a one out of two for being partly addressed.

Android has a very similar test suite that can monitor memory, cpu, gpu and network use. However, Android networking monitor only tracks the utilization of the an apps network connectivity. It does not contain any diagnostics on which port or what type of connection each of the connections are. Outside of IDE test suites, Google actually does provide a quite usable network traffic security testing tool called nogotofail[37]. Its purpose is literally to test for weak spots in TLS/SSL connections. However, nogotofail was only discovered as a result of consulting Steins paper. This testing tool is not readily linked in the Android documentation. As a result, Android will only receive a half score on its testing being addressed.

4.2.2 Certificate Pinning

Documentation

Android coverage for certificate pinning is slightly stronger than that of iOS because its documentation has a page that represents a certificate pinning landing page, while iOS does not. Therefor Android receives a full score, while iOS will receive two out of three points.

Table 4.5: iOS and Android Certificate Pinning Coverage

	iOS	Android
Secure Comms home page	✓	✓
Certificate Pinning home page		✓
Certificate Pinning Solution	✓	✓
Library docs	✓	✓

Certificate pinning is often considered to be a somewhat advanced security mechanism to be implemented. It requires the developer to understand TLS/SSL communication, as well as the certificate chain of trust quite well prior to implementation. The most current iOS implementation still reflects such complexity in its implementation, as it requires a series of case statements pertaining to complex API calls. Android on the other hand recently made certificate pinning dead simple in their Android Nougat release of Network Security configuration. Because of the increased certificate pinning implementation simplicity, Android has made sure to advertise. Upon its release, Android made a very clear landing page essentially on the Security homepage that discusses all of the unique things one can now do with the certificate chain of trust. Apples iOS on the other hand still has its certificate pinning documentation deeply hidden in its documentation, not even offering a landing page for the mechanism.

As alluded to previously, certificate pinning was heavily simplified in Androids Nougat release. As a result they have published comprehensive documentation associated with the security mechanism. All pages in the Android pinning document set are intuitively linked and like their HTTPS documentation, their pinning home page also contains the full solution for implementing certificate pinning into an Android app. Because of this Android receives a full score on its structure.

Figure 4.9: iOS Certificate Pinning Documentation Structure

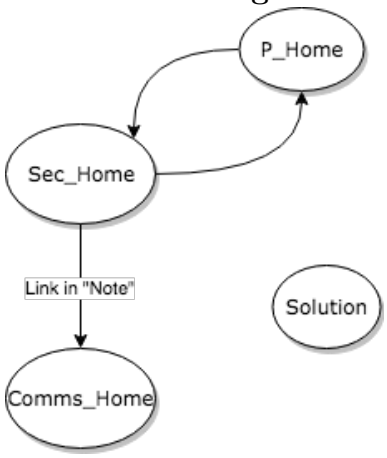
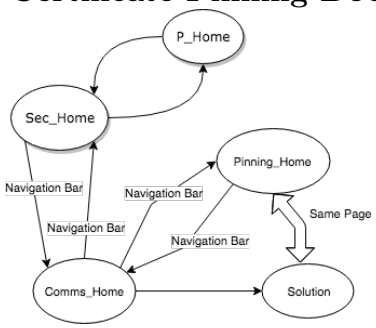


Figure 4.10: Android Certificate Pinning Documentation Structure



In contrast, iOS is far behind Android in terms of their documentation support for certificate pinning. They are not only missing a landing page for certificate pinning that will create awareness for the security mechanism, their solution is not even directly linked to the main network of certificate pinning associated pages. As a result, Apple receives a score of zero on structure. They are not only missing a

document from the graph but their solution is also indirectly linked to the other important documents of the document set.

Androids addition of the network security configuration in Android Nougat was in direct response to rising popular doubt in OS supported certificates[38]. They wanted to make it simple for programmers to implement the typically complex custom certificate settings. And they did just that. Certificate pinning on Android can now be achieved through declarative entries into an xml file that will link with the projects AndroidManifest.xml file. The manifest file describes essential information about the app to the Android build tools, the operating system and Google play. By linking the network security configuration for certificate pinning to the manifest file, the app will compile and ship with the defined certificate settings. The full xml file solution is located on the certificate pinning landing page and can be copy and pasted into an xml file. All that needs to be changed beyond the copy paste is the hash specified as the certificate and/or public key, and how the network security settings will link to the manifest file.

The iOS solution evaluated in this section is found in full in the legacy documentation database. This solution is secure and correct, and written in objective-C. However, iOS is docked one point because the complimenting instruction is overly verbose and uses vocabulary that is poorly previewed. Without a background understanding of the related APIs this documentation would be quite difficult to follow.

One final observation of both the iOS and Android solutions that both documentation sets discuss how to incorporate and existing form of a certificate into the application. However they do not discuss how to create that form. For example, the Android solution uses the hash of a public key and the iOS solution uses a DER encoded certificate. But neither documentation shows how to get that information. Through consultation of the OWASP certificate pinning documentation[19], it was

found that the developer must actually use the command line tool openssl to get the correct information. This is not at all documented in either platform documentation. As a result both platforms are docked one point for being incomplete. Therefore iOS receive one out of three points for their solution and Android receive two out of three points.

Please note that the solution that will be evaluated in the implementation section is one that was created as a byproduct from reading through the fragmented code in iOSs newer documentation set and the previously evaluated Objective-C code. As stated in the implementation section, we planned to present convertible Objective-C solutions in Swift as an added convenience to the reader. To do this we followed the objective-C solution as a guide. Then using the URL Loading system documentation, which was identified as the home for secure communication in iOS documentation, we identified the needed Swift controls. Through this approach we were able to create a cleaner Swift implementation very similar to the objective-C code was created.

Implementation

The certificate pinning implementation for iOS relies heavily on being implemented with a URLSession delegate. This is great example as to why the HTTPS section evaluated the the delegate implementation instead of the shared instance implementation. In this section we will simply build upon the connection that we established in the prior HTTPS section. So please refer back to the previous section to refresh yourself with the HTTPS implementation.

In order to extend the previously identified HTTPS connection to support certificate pinning one must add one more delegate function and one more field. Both additions are shown below.

Listing 4.12: Array that contains data for pinned certificates (iOS - Swift)

```
1 var certificates: [Data] = {
```

```
2     let url = Bundle.main.url(forResource: "example", withExtension:
          "cer")!
3      // Note the bang, because we want to crash the program if it
          does not load the cert
4      let data = try! Data(contentsOf: url)
5      return [data]
6 }()
```

First and most importantly the developer must specify an array of data that contains all of the certificates that they would like to have pinned. One should make sure to use an array because it is considered best practice to pin multiple certificates when implementing certificate pinning[19]. An example of why one should use multiple certificates is that certificates often are only valid over a certain time period. If a certificate becomes outdated while users have the app downloaded, the app will cease to connect to the server. As a result a costly rebundle and re ship would be required simply to update the certificate. By using multiple certificates one can more effectively cover such a misstep. Once the certificate is loaded into the array the only remaining step is adding the didrecieve challenge delegate method to the previously created HTTPS implementation shown below.

Listing 4.13: Delegate function to handle Challenge (iOS - Swift)

```
1 func urlSession(_ session: URLSession, didReceive challenge:
      URLAuthenticationChallenge, completionHandler: @escaping (
      URLSession.AuthChallengeDisposition, URLCredential?) -> Void) {
2      if let  trust = challenge.protectionSpace.serverTrust,
3              SecTrustGetCertificateCount(trust) > 0,
4              let certificate = SecTrustGetCertificateAtIndex(trust,
                   0) {
5
6                  let data = SecCertificateCopyData(certificate) as
                       Data
```

```
7                   if certificates.contains(data) {
8                       completionHandler(.useCredential, URLCredential(
                            trust: trust))
9                       return
10                  }
11              }
12      completionHandler(.cancelAuthenticationChallenge, nil)
13  }
```

The core idea in understanding how iOS handles certificate pinning relates to the URLAuthenticationChallenge class. Apple defines this class as the challenge from a server requiring authentication from the client. Depending on the type of challenge being sent, the client must respond appropriately. In the case of certificate pinning it gets slightly confusing because the app is not responding to the challenge sent from the server. The app is making use of the data that comes attached to the challenge in order to verify that the servers certificate matches the certificate the the app has pinned in its bundle. This data associated with the certificates is located in what Apple refers to as the core of the challenge; the protection space. The protection space is where the developer will find things like the type of authentication being required from the server, networking info, and most importantly for our purposes, the server trust object which contains all of the certificate info. In the above listing, 4.13, one can see that after verifying information about the servers trust, the implementation directly checks its certificates field to see if it contains the certificate data being sent by the server (at line 147). If the certificates array contains the same certificate data that the server sent, we verify the connection by passing the .useCredential to the received completion handler. If it is discovered that the certificate actually is not in the pinned set, we cancel the connection by passing .cancelAuthenticationChallenge to the completion handler.

As stated in the documentation section, Android has made its implementation of certificate pinning very simple through its addition of network security configuration in Android Nougat. Androids goal in implementing network security configuration was to make it simple to customize network security settings in a safe, declarative, configuration file without modifying any app code. With this implementation an Android developer can add custom anchor certificates, opt out of clear-text traffic (essentially same thing as apples ATS), and pin certificates. And this can all be done by simply adding or editing an xml file called network_security_config.xml, and then linking it to the Android_manifest.xml file. Listing 4.14 shows how to declare the use of certificate pinning in the network_security_config.xml, and listing 4.15 shows how to link the certificate pinning settings.

Listing 4.14: network_security_config.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <network-security-config>
3      <domain-config>
4          <domain includeSubdomains="true">example.com</domain>
5          <pin-set expiration="2018-01-01">
6              <pin digest="SHA-256">7
                   HIpactkIAq2Y49orFOOQKurWxmmSFZhBCoQYcRhJ3Y=</pin>
7              <!-- backup pin -->
8              <pin digest="SHA-256">fwza0LRMXouZHRC8Ei+4PyuldPDcf3UKgO
                   /04cDM1oE=</pin>
9          </pin-set>
10     </domain-config>
11 </network-security-config>
```

Listing 4.15: Linking the Android_manifest.xml to the network_security_config.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest ... >
```

```
3     <application android:networkSecurityConfig="@xml/
          network_security_config"
4                     ... >
5          ...
6     </application>
7 </manifest>
```

The Android configuration for certificate pinning is shockingly simple; the only choice that the developer must make is which digest to put in the xml file. As a result Android certificate pinning receives full marks on Abstraction. Considering the complexity of certificate pinning the iOS implementation is decently abstracted. Its only downfall is that its configuration requires the developer to create the verification logic. As a result, iOS certificate pinning will be docked 1 point in their level of abstraction score. Once again however it is worth noting that the lower abstraction for iOS enables more flexibility for the developer in implementing certificate pinning. So if the developer is very used to the more complex implementation of iOS, the API may serve them better.

As discussed in the solution evaluation, both the iOS and Android documentation neglect discussion of what information should be pinned, as well as how to get it. In contrast, when OWASP documentation introduces certificate pinning [19], one of the first things they mention is that there are a number of different ways to pin a certificate. This makes the implementation a lot more transparent. It was found through this OWASP documentation one should use the openssl command line tools to gather such information and add them to the app bundle. Such a task is identified to be a considerable amount of overhead work with little to no direction from the respective platforms. As a result both Android and iOS are docked one point on their minimality and are scored two out of three points.

We evaluate the robustness for Android to be 2 out of 2. Since the solution is

clearly provided the only error that is likely to occur is if the developer implements with the incorrect public key hash. However if this occurs the connection will likely just deny all incoming requests; unless by some miracle a certificate randomly matches the incorrectly entered value, which is highly unlikely. The biggest issue that could occur in implementing certificate pinning for iOS is missimplementing the logic in the challenge handler. This is a likely possibility because some of the verification constants are ambiguously named. For example the solution we identified online makes use of the constants kSecTrustResultUnspecified and kSecTrustResultProceed. When both of these evaluate to true in comparison with the trust object being evaluated, the connection is supposed to be approved. Since such an error is moderately likely and could potentially render all connections valid, the iOS implementation receives a zero in robustness.

Testing

As mentioned in the HTTPS, Google provides the network security testing tool called Nogotofail[37]. It is meant to be used as a tool to identify weak spots in TLS/SSL communication. As a result it not only can identify whether HTTPS is being used, but can also verify that the certificate chain of trust is being appropriately used. Unfortunately, the tool does not allow the testing of specific certificates. So it is quite difficult to verify that the pinned certificate is being used. As a result Android will receive a half score on its certificate pinning being tested. In contrast, iOS does not provide any testing resources for certificate pinning, so iOS receives a zero on its certificate pinning testing being addressed.

4.3 Insecure Authentication

While there are many ways that a developer can authenticate their user; most are not directly supported by the iOS and Android platforms. As a result, this section

will focus on device credential authentication because it is considered one of the more secure practices in app development for secure authentication. And because both iOS and Android take the most responsibility for providing these security mechanisms. In this sections we will be evaluating how usable the APIs are for both passode and biometric authentication.

4.3.1 Device Credential Authentication

Documentation

Apples iOS provides a document that satisfiably represents each of the documents in the document set. Therefore iOS achieves a full score on their coverage. Android on the other hand is missing a page that could represent that of a secure authentication landing page.

Table 4.6: iOS and Android Device Authentication Coverage

	iOS	Android
Secure Authentication home page	✓	
Device Authentication home page	✓	✓
Device Authentication Solution	✓	✓
Library docs	✓	✓

Something that will stand out about the structure of the iOS documentation shown below is that the the vulnerability page associated with secure authentication is disconnected from the remainder of the documents in the document set. As a result iOS is docked a point and receives a 2 out of 3 on its documentation structure.

Figure 4.11: iOS Device Authentication Documentation Structure

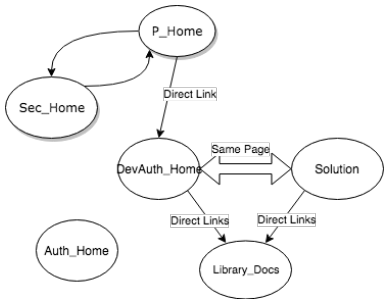
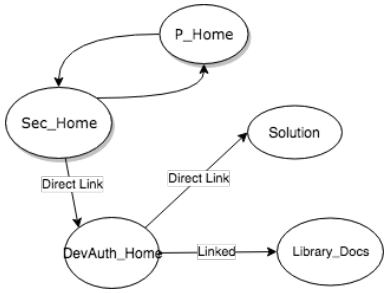


Figure 4.12: Android Device Authentication Documentation Structure



One more thing to note is that the iOS structure takes on an odd formation in that the security mechanism homepage is directly linked to the programming home page but not the security home page. This is a bit of a red flag because the iOS programming homepage is difficult to navigate unless you know exactly what you are looking for. Therefore, unless the programmer is specifically looking for local authentication documentation, the iOS documentation will not create any awareness for its importance to the developer. This however is not enough to bump the structure down any further.

The Android structure will receive a 2 out of 3 on its structure even though all available pages are directly linked. As we stated in the implementation section, a given security mechanism will lose a point if a page from its document set is missing. Such a case indicates that the platform is not providing sufficient awareness for the given security mechanism implementation to the developer. This issue is clearly seen through the Android documentation since the only way one will find device credential

authentication is by clicking directly on its security mechanism home page. There is no page that indicates the need for device credential authentication.

Finally, iOS will receive three out of three points on its solution. The solution is conveniently embedded in the device credential home page that we identified. The page is up-to-date, in the iOS developer guide, secure, and has complete and clear instruction associated with it.

The Android solution is of project format so it is indirectly linked to the device credential authentication homepage. The actual code in the project is up-to-date and secure, it also has all necessary instructions in the format of comments throughout the project. However, one will quickly realize that between both the directions with the solution and the security mechanism homepage, there are only directions explaining how to implement fingerprint authentication, but no passcode authentication. Whereas iOS conveniently explained both on the same page. As a result Android receives a 2 out of 3 on its solutions.

Implementation

The implementation of iOS device credential authentication relies on the localAuthentication framework. This framework is what allows for authentication from both passphrase and biometrics. And conveniently one can implement both with nearly the exact same code. The below code shows how to implement biometric(both faceId and fingerprint). As a result we will first explain how to implement with biometrics. Then once that is done we will explain the changes necessary for implementing with only passcode.

To implement biometric device credential authentication with Swift, one must first initialize an LAContext object[39]. The LAContext is the cornerstone of this implementation because it essentially coordinates the entire authentication process. If you take a look at the code snippet below, one must first use the LAContext

object to verify that all necessary biometric hardware is available. If the hardware is not available, this evaluation will provide the programmer with useful feedback. After that, the LAContext object is used in the actual evaluation of the face or touch id. This is done with the .evaluatePolicy() method where it evaluates the submitted authentication data based on the authentication policy. After the biometric data is submitted and evaluated it will pass a boolean referred to as success to the .evaluatePolicy()s completion handler. If the authentication was successful the success boolean will have value of true, otherwise it will be false. The code snippet then clearly identifies where to place the code associated with a successful authentication and a failed authentication.

One important thing to note with face id, is that in order to use it one is required to include the NSFaceIDUsageDescription key in the given apps Info.plist file. If that key is not present, authorization requests will likely fail. This is the only part of the solution that is not included in the security mechanism homepage. However it is clearly noted in the LAContext documentation.

Listing 4.16: Biometric and passcode device credential Authentication (iOS - Swift)

```
1 let myContext = LAContext()
2 let myLocalizedReasonString = <..String explaning why app needs
     authentication..>
3
4 var authError: NSError?
5 if #available(iOS 8.0, macOS 10.12.1, *) {
6     if myContext.canEvaluatePolicy(.
         deviceOwnerAuthenticationWithBiometrics, error: &authError) {
7         myContext.evaluatePolicy(.
             deviceOwnerAuthenticationWithBiometrics, localizedReason:
              myLocalizedReasonString)
8         { success, evaluateError in
```

```
9               if success {
10                  // User authenticated successfully, take appropriate
                        action
11              } else {
12                  // User did not authenticate successfully, look at
                        error and take appropriate action
13              }
14          }
15      } else {
16          // Could not evaluate policy; look at authError and present
                an appropriate message to user
17      }
18  } else {
19      // Fallback on earlier versions
20  }
```

The process of implementing device credential authentication with passcode is almost identical to the biometric implementation. The only difference is what we specify as the LAPolicy argument in the evaluatePolicy() function. In the above example it can be seen that the .deviceOwnerAuthenticationWithBiometrics case of the LAPolicy enumeration is specified. That constant indicates that when evaluating to authenticate the user, the user must be authenticated through biometry. But if we change the LAPolicy to .deviceOwnerAuthentication, the only other constant in the enumeration, the authentication will now require passphrase authentication.

The iOS implementation for device credential authentication is overwhelmingly simple considering the complexity of local authentication. The LAContext object abstracts away all failed attempts, all UI creation, and all hardware interaction. And the developer is left with a simple double conditional that clearly lays out the resulting logic flow. Plus the switch between biometric authentication and passphrase

authentication is as easy as changing a constant. As a result iOS undoubtedly receives a full score on abstraction. The implementation is also extremely modular. There is almost no overhead in its implementation. As a result it will receive a full score on minimality as well.

As mentioned in the abstraction evaluation, there are almost no decisions that need to be made by the developer. The only one that stands out is the misspecification of the LAPolicy constant in the evaluatePolicy method. However, there are only two options; one for biometric and one for passcode, which will both maintain some level of security. The only other identifiable error is the misinterpretation of the logic flow in the evaluatePolicy completion handler. However, the comments included in the code snippet are directly taken from the iOS solution. As a result I feel that it is quite unlikely the developer will misinterpret the logic flow. As a result, iOS will receive a full score on Robustness as well.

Unlike the iOS implementation, the Android implementation for fingerprint authentication and password authentication vary greatly. As a result we will discuss both implementations and grade the device credential implementation based on the worst score.

In order to implement fingerprint authentication in Android the developer must specify permission in the Android_manifest.xml file the line identified in figure 4.17. This is necessary because when an app needs access to certain hardware on a device, a line like this in the Android_manifest.xml is what actually makes the request to use the given hardware. Without this, none of the code in the project would be able to make use of the requested hardware.

Listing 4.17: Android_Manifest.xml for fingerprint authentication (Android)

```
1 <uses-permission android:name="android.permission.USE_FINGERPRINT"/>
```

Once the developer has given the app access to the fingerprint scanner, she has two remaining major steps. First the developer must write the appropriate checks to verify that the fingerprint scanner can be used. These checks are pictured in Figure **??**. Second, the developer must create a class that extends the FingerprintManager.AuthenticationCallback class, and then override its callback methods. This is seen in figure refFigure 8: Fingerprint Authentication Class (Android - Java).

Listing 4.18: Logic prior to authenticating a fingerprint (Android - Java)

```
1  protected void onCreate(Bundle savedInstanceState) {
2      super.onCreate(savedInstanceState);
3      setContentView(R.layout.activity_main);
4      KeyguardManager keyguardManager = (KeyguardManager)
           getSystemService(KEYGUARD_SERVICE);
5      FingerprintManager fingerprintManager = (FingerprintManager)
           getSystemService(FINGERPRINT_SERVICE);
6
7      TextView errorText = (TextView) findViewById(R.id.error_message)
           ;
8
9      // Check whether the device has a Fingerprint sensor.
10     if (!fingerprintManager.isHardwareDetected()) {
11         errorText.setText(getResources().getString(R.string.
               fingerprint_not_exist));
12     } else {
13         // Checks whether fingerprint permission is set on manifest
14         if (ActivityCompat.checkSelfPermission(this, Manifest.
               permission.USE_FINGERPRINT) != PackageManager.
               PERMISSION_GRANTED) {
15             errorText.setText(getResources().getString(R.string.
                   fingerprint_not_enabled));
16         } else {
17             // Check whether at least one fingerprint is registered
```

```
                on your device
18              if (!fingerprintManager.hasEnrolledFingerprints()) {
19                  errorText.setText(getResources().getString(R.string.
                        fingerprint_not_registered));
20              } else {
21                  // Checks whether lock screen security is enabled or
                        not
22                  if (!keyguardManager.isKeyguardSecure()) {
23                      errorText.setText(getResources().getString(R.
                            string.lock_screen_setting_disabled));
24                  } else {
25                      FingerPrintHandler helper = new
                            FingerPrintHandler(this, errorText);//Set
                            Fingerprint Handler class
26                      helper.startAuth(fingerprintManager);//now start
                            authentication process
27                  }
28              }
29          }
30      }
31  }
```

Listing 4.19: Fingerprint Authentication Class (Android - Java)

```
1   public class FingerPrintHandler extends FingerprintManager.
        AuthenticationCallback {
2       private Context context;
3       private TextView errorText;
4
5       public FingerPrintHandler(Context mContext, TextView errorText)
            {
6           this.errorText=errorText;
7           context = mContext;
```

```
8       }
9
10      public void startAuth(FingerprintManager manager) {
11          CancellationSignal cancellationSignal = new
                CancellationSignal();
12
13          // check permission of object requesting to authenticate
                with fingerprint
14          if (ActivityCompat.checkSelfPermission(context, Manifest.
                permission.USE_FINGERPRINT) != PackageManager.
                PERMISSION_GRANTED) {
15              return;
16          }
17          manager.authenticate(null, cancellationSignal, 0, this, null
                );
18      }
19
20      /*  Method will call on Fingerprint Auth Succeeded  */
21      @Override
22      public void onAuthenticationSucceeded(FingerprintManager.
            AuthenticationResult result) {
23          Log.d("Authentication", "Fingerprint␣Authentication␣
                successful.");
24          onAuthSuccess();
25      }
26
27      /*  Method will call on Fingerprint Auth Failed  */
28      @Override
29      public void onAuthenticationFailed() {
30          this.update("Fingerprint␣Authentication␣failed.␣Please␣try␣
                again.");
31          Log.d("myTag", "Auth␣failed");
```

```
32      }
33
34      /*  Method will call on Fingerprint Auth Error  */
35      @Override
36      public void onAuthenticationError(int errMsgId, CharSequence
           errString) {
37          this.update("Fingerprint␣Authentication␣error\n" + errString
               );
38      }
39
40      /*  Method will call on Fingerprint Auth have some help  */
41      @Override
42      public void onAuthenticationHelp(int helpMsgId, CharSequence
           helpString) {
43          this.update("Fingerprint␣Authentication␣help\n" + helpString
               );
44      }
45
46      /*  Trigger this method on FingerPrint get Success  */
47      private void onAuthSuccess() {
48          this.update("Fingerprint␣Authentication␣Succeeded!!!.␣App␣
               use␣is␣now␣disabled");
49          //context.startActivity(new Intent(context, WelcomeActivity.
               class));
50          ////((AppCompatActivity) context).finish();
51      }
52
53      /*  Method to update Error text message on Auth failed  */
54      public void update(String e) {
55          errorText.setText(e);
56      }
57  }
```

By taking a look at figure 4.18, one can see that prior to the actual authentication of biometric or passcode data, a number of cases must be satisfied using the fingerprintManager object and the keyguardManager object. It must be verified that the device has a fingerprint scanner, that the app has fingerprint permission, that there is a fingerprint enrolled on the device, and finally that lock screen security is enabled. Only once all of those things are verified can an of an authentication be called.

As stated before, unfortunately the fingerprint.authenticationcallback class must be manually implemented by the developer. Essentially this class handles the outcome of any fingerprint authentication process. For example it has a callback method called onAuthenticationSucceeded() which is called when the fingerprint matches one managed by the fingerprintManager. Likewise, the onAuthenticationFailed() is called when the fingerprint manager evaluates there to be no existing fingerprint associated with the submitted one. It is important to note that this code only encompasses the authentication of a fingerprint. The UI still must be manually created or taken from the dialog project.

Listing 4.20: Device Credential Authentication with Passcode (Android - Java)

```
1  private static final int LOCK_REQUEST_CODE = 221;
2  private static final int SECURITY_SETTING_REQUEST_CODE = 233;
3
4  // method to authenticate app
5  // Keygaurd manager is used to unlock keyboard, and in this case
       retrieve intent for password authentication
6  // show password authentication dialog
7  private void authenticateApp() {
8      KeyguardManager keyguardManager = (KeyguardManager)
           getSystemService(KEYGUARD_SERVICE);
9
10     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
```

```
11
12          Intent i = keyguardManager.
                createConfirmDeviceCredentialIntent(getResources().
                getString(R.string.unlock),
13                  getResources().getString(R.string.confirm_pattern));
14          try {
15              startActivityForResult(i, LOCK_REQUEST_CODE);
16          } catch (Exception e) {
17              Intent intent = new Intent(Settings.
                    ACTION_SECURITY_SETTINGS);
18              try {
19                  startActivityForResult(intent,
                        SECURITY_SETTING_REQUEST_CODE);
20              } catch (Exception ex) {
21                  textView.setText(getResources().getString(R.string.
                        setting_label));
22              }
23          }
24      }
25  }
26
27  @Override
28  protected void onActivityResult(int requestCode, int resultCode,
        Intent data) {
29      super.onActivityResult(requestCode, resultCode, data);
30      switch (requestCode) {
31          case LOCK_REQUEST_CODE:
32              if (resultCode == RESULT_OK) {
33                  textView.setText(getResources().getString(R.string.
                        unlock_success));
34              } else {
35                  textView.setText(getResources().getString(R.string.
```

```
                    unlock_failed));
36              }
37              break;
38          case SECURITY_SETTING_REQUEST_CODE:
39              if (isDeviceSecure()) {
40                  Toast.makeText(this, getResources().getString(R.
                        string.device_is_secure), Toast.LENGTH_SHORT).
                        show();
41                  authenticateApp();
42              } else {
43                  textView.setText(getResources().getString(R.string.
                        security_device_cancelled));
44              }
45              break;
46      }
47  }
48
49  //method to return whether device has screen lock enabled or not
50  private boolean isDeviceSecure() {
51      KeyguardManager keyguardManager = (KeyguardManager)
            getSystemService(KEYGUARD_SERVICE);
52      return Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN
            && keyguardManager.isKeyguardSecure();
53  }
```

The Android implementation for passcode authentication requires the developer to make use of the keygaurd manager to get the .createConfirmDeviceCredentialIntent() intent. By overriding the onActivityResult() method and then immediately calling its parent method, the UI for password authentication is taken care of for the developer. Then the result of the passcode authentication is returned back to the overridden method and it can be evaluated to see if the password was correct. This is a much

cleaner implementation compared to the fingerprint scanner because the intent takes care of the UI for us.

The abstraction of the fingerprint implementation is quite good. The logic of the outcomes are clearly defined by callback functions, so there are no issue in mis evaluating authentication outcomes. Android will however lose one point on its abstraction because of the logic in its passcode authentication intent call. The LOCK_REQUEST_CODE and SECURITY$_{S}ETTING_{R}EQUEST_{C}ODEaresomewhatconfusingl$

Android fingerprint authentication requires the developer to work with a cipher and generate keys if they want to keep track of when a user adds or changes their fingerprint. Such logic is actually quite necessary if you want to create a secure app. Please note such logic was left out in the above code snippets because the purpose of this section is to evaluate the usability of authentication. However, by taking a look at the identified Android fingerprint sample project[40] that we identified as the solution for device credential authentication, one will see how much work must go into setting up the cryptography tools. Since this overhead is far greater than that of the actual configuration of the security mechanism, Android will receive a score of zero on its minimality.

Finally Android will receive a one out of three on its robustness. It was decided that there are no likely mistaken decisions that would undermine the security of the mechanism or the underlying security. However the code associated with both the fingerprint authentication and password authentication is quite verbose. As a result, if the developer does not fully understand each line of code in its implementation, a misimplementation is plausible, and could easily have an undermining effect on the device credential authentication. Through consultation with the robustness risk matrix, Android thereby receives a one out of three on robustness.

Testing

Local authentication testing is overall quite trivial for both iOS and Android. This is because the only real way to test device credential authentication is by manually testing that the implementation is verifying only the correct passphrase and/or biometric input. This must be done on a real device. But if it verifies correctly on the real device, the implementation can be considered correct. As a result both iOS and Android will receive an N/A since there is no real need for them to address much more on how to test the implementation.

Chapter 5

DISCUSSION

In the first part of this chapter, we will discuss the overall results of our study. Following that we identify some thematic issues that the each platform could improve upon. In the second half of the chapter we will bring the spotlight onto the evaluation framework used in this paper. In that part of the section we will do two things. We will compare how our results to Steins results in the previous iteration of this work and we will discuss limitations and improvements that pertain to the framework used in this paper.

5.1 Evaluation of Results

Table 5.3 shows a compacted summary of the results of this papers evaluation of iOS and Android usability. There are two tables, the first is the iOS evaluation and the second is the Android evaluation. Please feel free to refer back to this table as you read through the results summary.

Table 5.1: iOS Usability Results

		Encryption	MAC	Key Storage	HTTPS	Cert Pinning	Local Auth
Documentation	Coverage	3/3	2/3	3/3	3/3	2/3	3/3
	Structure	1/2	0/2	2/2	1/2	0/2	1/2
	Solution	3/3	3/3	3/3	3/3	1/3	3/3
Implementation	Abstraction	0/2	2/2	2/2	2/2	1/2	2/2
	Minimality	1/2	0/2	1/2	2/2	1/2	2/2
	Robustness	0/3	0/3	3/3	3/3	0/3	3/3
Testing	Addressed	2/2	1/2	N/A	1/2	0/2	N/A

Table 5.2: Android Usability Results

		Encryption	MAC	Key Storage	HTTPS	Cert Pinning	Local Auth
Documentation	Coverage	3/3	3/3	3/3	3/3	3/3	2/3
	Structure	1/2	1/2	2/2	2/2	2/2	1/2
	Solution	2/3	2/3	3/3	3/3	2/3	2/3
Implementation	Abstraction	1/2	2/2	2/2	2/2	2/2	1/2
	Minimality	2/2	2/2	2/2	2/2	1/2	0/2
	Robustness	0/3	1/3	3/3	1/3	3/3	1/3
Testing	Addressed	0/2	0/2	N/A	1/2	1/2	N/A

5.1.1 Summary of Results

Documentation

The results of this study prove that both iOS and Android put a lot of effort into creating usable documentation that sufficiently support the average developer in implementing security mechanisms. However, it is clear that the Android documentation is consistently okay to perfect scores, while the iOS documentation waivers from bad to terrible. We point this out because the degree of a platforms documentation weak spots is perhaps the best representation of their documentation effectiveness. An okay document set is something a developer can still make use of. A bad document set is simply useless. If a developer is unable to find an important page because the documentation structure is terrible, then they have nothing to confirm their thought process with. Whereas, if the structure is okay- with determination- the developer will likely discover what they need. Likewise, if the provided solution is terrible and hard to follow, the developer must create an implementation from scratch or by referencing a third party. And unfortunately, it is common for developers to copy over insecure code when consulting third party resources[41]. Whereas, if a developer has the option to consult an okay, but 100% secure solution provided by the platform, the likelihood of a secure implementation sky rockets.

The documentation that covered security mechanisms associated with insecure

data storage was one of the more interesting sections of this papers research. Key storage for iOS and Android had perfect documentation scores, as both acquired the highest scores across all documentation evaluations for their respective platform. However, each platform acquired some of their worst documentation scores for their cryptographic security mechanisms; encryption and MACs.

It was found that both of the cryptographic security mechanisms had okay to perfect coverage on both platforms, but their structure seemed to be a consistent issue. This was a result of the fact that the cryptographic libraries were typically linked through the respective platforms secondary databases (refer back to section 3.2.1 for reference). Both of these secondary databases consist of an older document set that has been around since early versions of iOS and Android. As a result they almost never reference the respective primary databases that typically contain the most recent and most useful documentation. Because of this, many documents in this section would not create awareness for other important pages in the set.

Likely the worst characteristic of the iOS cryptographic documentation was the location of its encryption and MAC solutions. Both solutions were located in a standalone Xcode project that was mentioned in the encryption homepage, but not directly linked. The project was referred to as crypto compatibility[29]. The only reason we were able to discover the CryptoCompatibility project was because we google searched its title. Worse yet if a developer were trying to implement only MACs, and not encryption, a solution likely would not have been found. This is because iOS MAC documentation simply does not mention a solution or even display a MAC homepage. The only reason the MAC solution was found was because we evaluated the encryption documentation first and found that cryptocompatability had solutions for both encryption and MACs.

The Android cryptographic documentation was similarly linked to its secondary

database; the Javadocs. Javadocs contains the Java Cryptography Architecture (JCA)[31] documentation, which has been Androids cryptographic framework since its early versions. As a result encryption and MAC homepages and solutions were located on this page. This is a slight downside in the Android documentation because upon initial investigation this page appears to lack encryption and MAC information for Android due to being a large page. It felt that such a characteristic would cause the average developer to overlook the useful information in the page if they are unaware that Android simply inherited the Java cryptographic library. However, if the developer understands this pages relevance, or if they take the time to read through the table of contents at the top, they will find that there is a quick link to the encryption and MAC portion of the page that also links to their respective solutions.

Documentation for secure communication was quite good for both iOS and Android with the exception of iOS certificate pinning. HTTPS is now a commonplace security mechanism that is almost expected in implementation[42], as result it is no surprise that both Android and iOS had near perfect scores in this category. However it was a surprise to see that Android had near perfect documentation on certificate pinning, while iOS certificate pinning ended up being the platforms worst documentation score. Consequently, the certificate pinning evaluation represented one of the largest cross platform variations in the documentation evaluation.

The reason for such variation across platform certificate pinning documentation becomes clear upon the realization of Androids recent addition of the network security configurations feature[43], which was released with Android Nougat. There have been numerous concerns in modern communication security in regards to the reliability of the certificate chain of trust used in SSL/TLS connections[44]. However, in the past implementing safeguards like certificate pinning to protect against such concerns has been quite difficult. The Android release of the network security configurations

feature was a direct response to this traditionally difficult task. The purpose of the network security configuration feature was to grant the developer control over their applications network security policy in a safe, declarative configuration file without modifying code. With its addition developers can now implement certificate pinning, customize trust anchors, and require encrypted traffic (similar to Apples ATS feature), all in a robust declarative manner.

Because Androids approach to implementing certificate pinning is so unique, yet simple, it is no surprise that they have made sure to create awareness for it. As a result, the Android certificate homepage is conveniently linked directly on the secure communication homepage. Plus, the Android documentation for certificate pinning has nearly everything a developer could possibly need to implement it. Apple on the other hand has a rather complex implementation that they seem to hide from the developer. It is indirectly linked in its secondary database, and is written in objective-C.

The evaluation of secure authentication documentation proved to favor iOS over Android documentation. Surprisingly, device credential authentication is the only documentation evaluation that actually favored iOS documentation over Android documentation for a given security mechanism. In this section we evaluated how well the documentation covered both biometric and password documentation. It was found that iOS had a simple API that allowed for the use of the same code to implement both password and biometric implementations. The only variation between the two implementations was a constant that had to be specified for the actual evaluation. In contrast, Android had two entirely different implementations for biometry and passcode. All things considered, we hypothesize that the reason the iOS documentation turned out to be more effective than Androids was a result of its APIs simplicity. Apples API was easy to explain for passcode and biometry at the same time, therefore it was easy to link to and provide information on it. Whereas

Androids implementation turned out to be quite complex and as a result,difficult to document effectively.

The documentation evaluation as a whole proved to be a very effective indication of how complex a given security mechanism would be. And vice versa, the complexity of the mechanism typically reflected a difficult to use document set. If the mechanism was recently built up to be easy to use, its information was easily decomposed into pages that were represented in our document set and were located in their primary database. Not to mention if the given mechanism was easy to use, the platform wanted to advertise their documentation on it. Great examples of extremely well-documented security mechanisms were key storage, HTTPS and iOS local device authentication. In each of these cases the respective platform had built up these mechanisms to be very intuitive and usable. As a result documentation was easy to write and there was incentive to advertise such low risk, high reward mechanisms on the developer website. In contrast, the cryptographic security mechanisms have not been built upon for years, they are not intuitive nor easy to use for the average programmer. As a result, new documentation would be difficult to write, and such documentation would not reflect well on the respective platform.

Implementation

Our findings for the implementation evaluation of both iOS and Android mirrored that of the documentation evaluation findings. Such a correlation seems to be a result of the implementation. This is likely because a well-abstracted, minimal and robust implementation is often one that is easy to talk about and therefore more likely to be documented well.

Security mechanisms related to secure data storage followed the above theme and mirrored the low scores of their documentation. The Key Storage implementation was perfect for Android, while the iOS implementation was perfect with the exception of

its minimality. This was because its Objective-C implementation requires some extra set up in order to pass data to a C API call. Like the documentation evaluation, cryptographic security mechanisms for both iOS and Android were pretty difficult to use. However, iOS stood out as awful APIs, while Android cryptographic APIs were just not that robust.

The iOS cryptographic security mechanisms (encryption and MACs) were both found in a project called cryptocompatability[29], in the iOS secondary database. This project was not linked to the other documents in the document set and was written for Mac OS in Objective-C and C. Please note that this code was purely the algorithms so they could be used in iOS development. The iOS encryption implementation received the worst score out of all security mechanism implementation evaluations. Its abstraction received a zero because in order to call the encryption API the developer has to pass 11 parameters. It received a one out of two for its minimality since the API required the developer to deal with pointers and memory allocation in C. And finally, its robustness received a zero because it was decided that the platforms cannot expect the average developer to be comfortable with C. As a result, it was determined that the amount of advanced C knowledge required for the implementation severely threatened the underlying security of the app. Plus, the encryption API likely would have received a zero on robustness even without the required C code simply because of the difficulty level of accurately specifying all 11 parameters.

The iOS MAC implementation was similar to the encryption implementation in regards to its minimality and robustness. The iOS MAC implementation, like the iOS encryption implementation, required too much C familiarity from the developer. As a result the minimality and the robustness were both given zeros. Please not that MAC minimality received a zero while encryption received a half score because the methodology specifies that the relative difference between overhead work and

configuration work is compared in calculating a minimality score.

Android MACs and encryption were similar to the iOS implementation in that they used the original Java Cryptography Architecture[31] like iOS uses their original commonCrypto library[30]. This is not a huge issue for Android because if a cryptographic library has proven to be secure there is no need to change it. However what was surprising was that they had not built upon their original libraries to make them more usable for the average developer. Aside from this point though, the Android implementation actually scored well compared to iOS. The only issue with it was that they still exposed a lot of technical detail directly related to the characteristics of encryption and MACs. As a result both received somewhat low robustness scores.

Like secure communication documentation, secure communication implementation was, for the most part, well-implemented. The only difficult to use implementation was the iOS certificate pinning implementation, which again reflected its poor documentation.

The iOS implementation for HTTPS scored slightly better than Android as a result of its robustness score. Rendering the Android implementation insecure can be done through a simple typo in writing the url as an http connection instead of an https connection. The iOS implementation on the other hand guards against that exact error by using what Apple refers to as App Transport Security (ATS). With it enabled, if a developer were try to communicate using http only, Xcode will throw an exception saying that the connection was insecure stopping the application from running.

The iOS implementation for certificate pinning evaluated to be quite poor. Its implementation required the developer to implement their own logic into a delegate function to handle certificate integrity. As a result it received a half score on its abstraction and robustness. Android certificate pinning on the other hand was likely

the most usable implementation of all security mechanisms evaluated in this study; all because of Androids release of network security configurations[43]. Its release made it so that the developer can do things like pin a certificate and customize trust anchor all through xml declaration. The implementation doesnt even require the developer to write a single line of code. Interestingly, it even allows for a setting that is similar to Apples ATS feature: through the network security configuration file the developer can decide to disable all clear text communication.

Finally, the implementation evaluation of device credential authentication found iOS to be an extremely usable API. A developer could use essentially the exact same code for passcode, fingerprint and face id authentication. The only difference between each implementation was the value of a constant specified when the actual evaluation API method was called. Secondly, the iOS implementation abstracted away all failed attempts and all of the associated UI; leaving only what to do after the success or failure of the authentication up to the developer. Android scored poorly because it had two very different implementations for fingerprint and biometric, and both were verbose and overly complicated. The fingerprint implementation specifically had an unnecessarily large amount of overhead work, as a result device credential authentication received a robustness score of zero.

Testing

Overall testing by both platforms was decently addressed. The only security mechanism that offered fully useful test resources was the iOS encryption mechanism because it came with adequate test data. Likewise iOS MAC testing was almost fully addressed, but it did not offer data that would allow one to test SHA-256, it only gave test data for SHA-1. The iOS test data for both encryption and MACs were bundled with their solutions in the Cryptocompatability[29] project. Android did not offer any testing resources, or even discuss testing in regards to their cryptographic

security mechanisms.

Testing tools are offered by both iOS and Android to help test secure connections. However iOSs test suite, Instruments[36], only partly tests HTTPS by checking the port number of connections. And it cannot be used to check certificate information. Google does offer a tool called nogotofail[37] that does in fact test both HTTPS and certificate verification; however this tool was only found through consultation of Steins thesis. It was never discovered in the exploration of Android documentation.

Finally, testing was deemed not applicable for key storage and device authentication as their functionality verification is intuitive. As a result both were marked with N/A on the results sheet.

5.1.2 Platform Usability Improvements

In this section we discuss how both Android and iOS can improve their security mechanisms to be more usable. The section is split up to discuss platform specific improvements in documentation, implementation, and testing separately.

Documentation

Apple and Android documentation on average received quite similar good to perfect scores on their documentation. However something to note is that iOS received the two only zero documentation scores for it structure of MAC and certificate pinning documentation. As a result we believe a major priority for Apple should be to improve the structure of those two document sets. This would mean updating their documentation to reflect the typical document set that we outlined, or something comparable. By doing so the mechanisms will not only improve in their information coverage, but the correct awareness within the documents will also be addressed, therefore making the information much easier for the developer to navigate. The current state of iOS MAC and certificate pinning structure makes it nearly impossible

for the developer to find the necessary background on either mechanism. As a result this is a major issue in regards to the iOS documentation.

One common issue between the two platforms is their use of two documentation databases; their developer guide (primary database) and secondary databases. We refer to the iOS secondary database as its legacy database[24], and the Android secondary database is Javadocs[45]. And we suggest that both iOS and Android update such pages to be located in the developer guide and directly link to other related pages.

Our evaluation found that Apple still relies quite heavily on their secondary database. Often times if one tries to google search a given security vulnerability or mechanism, the legacy database will return. Also a majority of their vulnerability home pages are in the legacy database. The issue with this is not that the information on these pages is too old to use, although for many code snippets their recency is questionable. The major issue is that the developer guide has duplicate pages that are often newer versions of these legacy pages and better connected to the other pages in the document set. While these older pages are not at all linked to any of the new documentation in the document set. As a result, it is easy for the developer to find themselves researching in a rabbit hole of somewhat old information in the legacy database when they could likely find concise, recent, well-structured documentation in the developer guide.

Androids use of Javadocs is an issue, but a lesser one compared to iOSs use of their legacy documentation. This is because the only time Androids use of their secondary database is required in this study is when one tries to implement cryptographic security mechanisms; encryption and MACs. It turns out that a majority of Android cryptography documentation is found in the original Java Cryptography Architecture(JCA)[31] documentation. Such documentation is actually quite useful

and as recent as it needs to be. However, the document itself is far too large, containing a ton of unrelated cryptography information. It is also formatted in a way that an Android developer would not expect. When we first found the page, we almost instinctively left it because it was an entirely different format and did not feel like the right place for the information we were looking for. Because of this we suggest that Android update the information on the JCA documentation, so that it is broken up and rewritten in the Android documentation format. This way its different sections can be directly linked across the Android documentation, which will then contain only Android specific information. Such structure would make the cryptography documentation alot more intuitive, and therefore more usable.

Implementation

The iOS cryptography security mechanisms were the most difficult to implement in this paper. Both implementations have not been built on since the creation of commonCrypto[30], which handles symmetric asymmetric encryption and hash based authentication. This abstraction is an Objective-C library that is heavily based on C API calls to coreCrypto[25], an implementation of low level C cryptographic primitives. As a result the use of commonCrypto turned out to require the developer to be well versed in C programming. In our implementation it was found that for encryption you not only had to be familiar with C syntax, but also be comfortable with memory allocation, pointers, and casting. Something that the platform should not expect the average developer to implement. This not only made the APIs extremely difficult to configure, but also made them insufficiently robust. Our suggestion to Apple is to further abstract their commonCrypto library so that pure objective-C or swift can be used to carry out encryption or the creation of MACs.

The Android cryptographic implementations were not nearly as unstable as the iOS implementations. The only issue was that they revealed a lot of the cryptographic

detail to the developer. For example, their encryption implementation required all developers to choose the algorithm, mode, and padding for their implementation. In reality this is might actually be very convenient for the experienced programmer. But for the average programmer, the security mechanism would be more usable if the implementation were by default secure with no parameters. But then it could be customized to a different configuration if the developer really knew their stuff. If this were the case it would be simple to use for a programmer with no encryption experience. But still offer flexibility for the experienced programmer.

The implementations for for Key Storage and HTTPS were all quite good. Our only suggestion is that Android may want to imitate iOS in their implementation of ATS. Adding a tool to their IDE to protect against clear text traffic by default would be trivial to implement, and would thoroughly improve their HTTPS implementation robustness.

Certificate pinning and device credential authentication were interesting security mechanism evaluations because these were the only two implementations that received polar opposite evaluations across the two platforms. Android certificate pinning is fantastic as a result of its network security configuration[43] release, while the iOS implementation is quite poor. And the iOS device credential authentication is phenomenally abstracted to a point where both password and biometric authentication can be easily implemented in almost the exact same way. While the Android device credential authentication for passcode and biometric authentication are very different, and both are difficult to implement. Because of the variation our primary advice to improve implementation usability for both platforms is to simply take after their competitions implementation. Apple should try to add network security configurations to its info.plist so that it can imitate Androids network security configuration. And Android should further abstract its device authentication implementations so that it is simple to implement both password and biometric authentication in a similar

manner like iOS.

Testing

Overall testing was not very well addressed. Understandably there was nothing offered for secure authentication or key storage. However cryptographic mechanisms, HTTPS, and certificate pinning should have complete complimentary testing resources. Apple did a good job with its encryption testing data, but it is clearly outdated when it comes to MACs as it only provides SHA-1 test data and not SHA-256. Android needs to take after iOS and supply test data for both encryption and MACs.

Secondly, both iOS and Android have internal test suites to their developer IDEs. Apple needs to build upon their Instruments[36] test suite to test for weak spots in the SSL/TLS protocol as well as some sort of verification system on which certificates are being used by the apps. Likewise, Google needs to integrate their SSL/TLS testing suite, nogotofail[37], into Android Studio. This would be a simple way for them to allow for testing of both HTTPS and the correct use of certificate pinning.

5.2 Comparison to a Past Evaluation

Below you will find the results from Steins paper. We chose to simplify his results into a similar table to that of ours so that one would be a able to see the primary similarities and differences between our results and his. One important thing to note is that we have actually removed four criteria from Steins work in ours. This is because we found that the methodology for evaluating said fields were not sound in their conclusions. Also one will notice that Steins framework had structure and awareness as criteria in the documentation evaluation. Whereas we combined those two fields into one criteria called structure. As a result in the below evaluation one will find the average of Steins structure and awareness evaluation. This average is then directly compared to the Structure score of this paper. Please note that the

below results are directly comparable with our result presented in section 5.1.

Table 5.3: Stein's iOS Usability Results

		Encryption	MAC	Key Storage	HTTPS	Cert Pinning	Local Auth
Documentation	Coverage	100%	75%	100%	75%	100%	100%
	Structure	AVG(25,25)=25%	AVG(25,25)=25%	AVG(100,100)=100	AVG(50,100)=75%	AVG(25,0)=13%	AVG(75,0)=38%
	Solution	75%	75%	100%	100%	75%	100%
Implementation	Abstraction	25%	25%	100%	100%	50%	75%
	Minimality	0%	0%	50%	100%	75%	100%
	Robustness	0%	0%	100%	100%	25%	75%
Testing	Addressed	100%	25%	N/A	75%	0%	N/A

Table 5.4: Stein's Android Usability Results

		Encryption	MAC	Key Storage	HTTPS	Cert Pinning	Local Auth
Documentation	Coverage	100%	100%	100%	75%	100%	50%
	Structure	AVG(75,50)=63%	AVG(100,0)=50%	AVG(100,100)=100	AVG(100,100)=100	AVG(100,50)=75%	AVG(50,0)=25%
	Solution	50%	75%	100%	100%	100%	100%
Implementation	Abstraction	50%	50%	75%	100%	75%	75%
	Minimality	100%	100%	100%	100%	75%	0%
	Robustness	0%	25%	100%	50%	50%	75%
Testing	Addressed	0%	0%	N/A	25%	50%	N/A

In comparing the results one should notice that there are four possible scoring levels in the evaluation results represented by percentages. Criteria scores were represented with green at 75-100%, yellow at 50-75%, orange at 25-50%, or red at 0-25%. Over the entire set of results there were only three scores that varied more than one scoring level across the two results. The first major variation was the iOS certificate pinning solution, where our results showed a 33% and Steins results gave a 75%. The second was iOS MAC abstraction, our results gave a 100% while Steins gave a 25%. The final major variation was Android Local Authentication Robustness; our evaluation gave a 33% while Steins gave a 75%. Other than these three variations however, the scores across the two results were quite similar; typically resulting in the same scores, or at most just one level off.

The documentation evaluation evaluated 3 criteria for 6 mechanisms across iOS

and Android. So in total 36 documentation criteria evaluations were made. When comparing our results with Steins there were only 10 deviations across the two sets of results. And only one result varied more than one level. The implementation evaluation in our paper evaluated 3 criteria for 6 mechanisms across iOS and Android as well; evaluating 36 criteria. When comparing these results with Steins there were just 12 variations, with only 2 deviations greater than two levels. Finally, there were only three variations across the two sets of testing results for the 12 testing criteria evaluated.

Overall one can see that the results as a whole scored quite similarly. One will notice that there is slight variation across the score but this is simply a result of slightly varied processes for each of the criteria proposed in the two papers. For example the scoring for the robustness matrix was slightly varied to accommodate fewer possible scores. Such slight variation in these processes is why there were variations of greater than one level in Android Local Authentication robustness, and iOS certificate pinning solution. Such a case occurred for a number of other criteria as well. Furthermore, security mechanisms have changed since Steins iteration. So there are also a number of criteria scores that are variations as a result of different input data as opposed to a result of the lack of effectiveness of the framework. For example Android certificate pinning has vastly change and as a result there are three variations across the results there that are simply a result of a new security mechanism. Because of this, overall we identify that this iteration of the work further verifies that the usability framework to be quite effective. Although not every single criteria exactly matches, overall the two frameworks conclude with the similar results. For example, both frameworks identify that the cryptography libraries for both iOS and Android are not very usable. While they both agree that HTTPS and and Key Storage APIs are quite usable. This however does not fully verify the framework. At this point we have evaluated usability of these mechanism using four different evaluators. In

order to more fully verify this framework it would be worthwhile to carry out this evaluation in a survey fashion that would allow for many more people to test our findings.

5.3 Evaluation of The Framework

The main goal of this paper was to assess how well iOS and Android support the average developer in the secure implementation of security mechanism. By doing so we felt that it would reveal insights into which platform better supported the creation of secure applications. In order to carry out this evaluation we created a framework that evaluated the documentation, implementation, and testing of a number of the most used and most important security mechanisms created by iOS and Android. Our framework was inspired by the usability evaluation framework proposed by Florian Stein[6], and built upon it through the consultation of the cognitive dimensions framework[11]. In this section we will evaluate the perceived effectiveness of the framework that was proposed in this paper. And where applicable, we will compare similarities and differences between Steins framework and ours.

The evaluation was in its entirety, carried out by hand. Alternatives to a manual evaluation were heavily researched prior to evaluation, but ultimately no alternative techniques proved to be effective enough. The only realistic alternative was the use of surveys with experienced mobile developers as its participants. However this was ultimately ruled out as a possibility because of time constraints and a lack of both iOS and Android developers. Another option considered was automated testing. However this idea was quickly discarded, the evaluations in this paper are in nature very qualitative, and automation would not be able to adequately capture many of the important nuances of usability.

A major goal in re-creating the evaluation framework was to improve the documen-

tation evaluation. We wanted to simplify it to be easy for the reader to understand and conceptualize. As a result we came up with the concept of a document set. Through actual evaluation of security mechanisms in the study it was realized that there were about five types of pages that every well- documented security mechanism had. We identified these five pages as our document set. The document set was a very important improvement to our documentation evaluation because it allowed us to quantify how well the respective platform covered all necessary information associated with the security mechanism. Secondly, with the document set, we now had a universal way to map the connections between the information associated with each security mechanism. As a result our structure criteria uniquely used a graph to represent two of Steins criteria: structure and awareness. It was felt that by presenting the document set in the form a graph the reader was able to intuitively understand the layout of the platform provided information. This technique also allowed for a more quantitative way of grading the structure and awareness that the documentation provided.

Our implementation evaluation was somewhat inherited from Steins framework proposal. The only difference was that we discarded one of his criterion because it turned out to be overly difficult to measure. The abstraction and minimality criteria allowed us to split the implementation into two workloads which made the usability evaluation quite intuitive. It was identified that the work done to implement a security mechanism could be broken up into one of two categories: (1)work directly associated with the configuration of the mechanism, and (2)work that was unrelated to the mechanism but necessary for its incorporation into a project. Using the criteria abstraction and minimality we were able to satisfiably measure both in a quantifiable way. Finally we measured how a error could affect a given security mechanism with the robustness score. The validity of this evaluation is still questionable because it relies on the assumption that the evaluator can identify all possible errors. However,

it was ultimately kept because it does offer a useful perspective on how each of the security mechanisms function. The technique of evaluating robustness though can no doubt be improved upon.

Limitations

One major limitation of the framework is its inherent subjectiveness. Because the entire evaluation is, for the most part, carried out through manual evaluation, the evaluators previous knowledge brings unavoidable evaluation bias. As an experienced programmer there were likely many confusing subtleties that were skimmed over as a result of comfort with the two platforms. If this paper could be evaluated through a survey a more objective set of results could be drawn.

Another limitation of this paper that was highlighted was the inclusion of the primary and secondary databases for iOS. Apple has what this paper refers to as a legacy database that a lot of its vital documentation are located in. These documents are informative, but often verbose and completely separate of their newer more informative developer guide. Often times throughout the evaluation these legacy documents took us down a rabbit hole of information that was not useful for actual implementation. While the developer guide provided concise well linked explanations for many of the security mechanism. As a result the use of legacy documents in a future work should affect the score of the platforms documentation scores in some way.

Also, the implementation evaluation is one that can always be improved upon. Overall we still approve of our implementation evaluation because the usability of a security mechanism is simply difficult to measure. And it is felt that our approach effectively decomposed each implementation into very useful criteria. But, if this study is carried out again, we would suggest further consulting the cognitive dimensions framework[11] to improve upon it. A majority of our time was spent improving on the documentation section in consultation with Greenes approach. The next natural

step in improving the framework would be to do the same for the implementation evaluation.

Finally, a major assumption of this paper is that it assumes the correct implementation for all security mechanisms by the respective platforms. The goal of this paper is the evaluation of each security mechanisms usability. Although this paper does test the basic functionality of each mechanism at the end of the study. Such testing is not enough to confirm that each implementation is fully functional. As a result, it is possible that this framework could be used to evaluate insecure security mechanisms and the difference would not be apparent.

Chapter 6

CONCLUSION

The purpose of this work is to compare the usability of security mechanisms created by iOS and Android. By doing this we expected to identify which platform best supported the average developer in creating secure applications. Thereby reflecting a very important characteristic of how secure each platform is. In order to evaluate the usability of security mechanisms we created an evaluation framework inspired by one created by Stein[6]. It was then built upon in consultation with the cognitive dimensions framework[11]; Steins original inspiration. We then used this framework to evaluate seven security mechanism that were identified as some of the most important and highly used in the OWASP top ten vulnerabilities list**??**.

It was identified that the number one vulnerability in mobile security according to OWASP is that developers are misusing security mechanisms and thereby rendering their applications insecure**??**. This fact is the primary inspiration for this paper because it brings to light an interesting fact. This fact being that the correct implementation of security mechanisms is heavily correlated to the support given by the respective platforms that built them. This chapter also outlines the main goals of this paper. (1)To compare the usability of seven of the most important security mechanisms created by iOS and Android. (2) Provide an evaluation framework that can be used to evaluate the usability of a given security mechanism. (3) To extend on the OWASP mission to aid the average developer in secure implementation of mobile applications.

Currently, the only paper directly related to the usability of security mechanisms is Steins paper; A Framework for the evaluation of Mobile Platform Support for implementing security Mechanisms[6]. In this section we also discussed past approaches

in assessing the security of security mechanisms. We also devoted a lot of content to research of usability. There are numerous ways to assess usability; automation, surveys, cognitive frameworks. In this section we discussed the benefits and negatives of each, and ultimately identified that the cognitive dimensions framework was best suited for the work being carried out in this study.

Once it was decided that our evaluation framework would be one based on the cognitive dimensions framework, we were able to propose our own evaluation framework specifically meant for evaluating security mechanisms. The security mechanism chosen for evaluation were taken directly from the OWASP top ten list. It was identified that the evaluation should be broken up into three main parts in order to reflect the software development cycle. As a result it was broken up into three sections that would evaluate the documentation, the implementation and the testing respectively of each security mechanism. Then within each subsection of the evaluation criteria we defined criteria that could be quantitatively scored in order to reflect how usable each evaluated security mechanism is.

Once the security mechanisms and the evaluation framework were identified the actual evaluation was reported on; the evaluation can be found in chapter 4. This chapter was broken up into three sections based on the security vulnerability that each of the security mechanisms defended against. The Insecure Data Storage section, section 4.1, evaluated the usability of the arbitrary encryption, MAC, and key storage mechanism. The insecure communication section, section 4.2, evaluated HTTPS and certificate pinning. And the insecure authentication section, section 4.3, evaluated device credential authentication.

A summary of our evaluation can be found in chapter 5. In this section we discussed the strong points of Android and iOS support, as well as their main weak spots. There were also a number of unexpected findings that were cited in this section. It was

found that overall both iOS and Android put a lot of effort into making well-designed security mechanisms with effective documentation and testing tools. However, on average Android security mechanism documentation and implementations were more usable.

Android documentation as a whole was comprehensive and well structured for nearly all security mechanisms evaluated in this study. It was almost entirely located in their developer guide so all necessary information was easily found and clearly outlined. iOS did a good job too, but wavered often because it required the reference to outdated documentation located in its secondary document database (secondary document database defined in section 3.2.1). A majority of the iOS pages that offered security vulnerability background were located in this older document set. Because of this, if a developer was only referencing the developer guide, they would likely not be made aware of the importance for many of the security mechanisms. Likewise, these older documents were never linked to the other more recent and useful documents that are in the developer guide. So developers referencing the older documents would likely miss out on some of the most useful information needed to implement certain security mechanisms.

iOS had extremely usable and well-designed implementations for key storage, HTTPS, and device credential authentication. However, their encryption, MAC and certificate pinning implementations were very difficult to use. For example, the iOS cryptography mechanisms (encryption and MACs) require the developer to write in C and use low level C APIs that should realistically be abstracted upon by now. Android on the other hand, only had one real weak spot- its device credential authentication. From a different perspective, however, Apple implementations may have been the more difficult mechanisms to implement. But they offered more flexibility in their implementation. As a result, some more experienced developers may prefer to work with iOS despite its lesser usability.

In the second half of chapter five we re-evaluated the effectiveness of the framework that this paper proposed. In this section we pointed out a few of the major downfalls in our proposed usability evaluation framework. The primary limitation that we identified was the subjective nature of our framework. It was identified that the act of evaluating usability had to be qualitative since usability is in nature very abstract. So despite its limitations we still felt our cognitive approach was the only effective way to evaluate usability. We also identified that the best way to improve upon this limitation would be to turn this study into a survey in order to compile many perspectives in the evaluation.

In summary, this study served as a proposal of a usability framework that could be used to evaluate the usability of mobile security mechanisms. It also made use of the framework in order to compare the usability of the iOS and Android platforms alike. By carrying out this study we found that on average, Android provided more usable documentation, and supplied implementations that made it much easier for the average developer to use. Despite iOSs typical increased complexity, it was identified that all security mechanisms did have supporting documentation and implementations, they were just more complicated. As a result, in some cases the iOS implementations may be preferable to more experienced programmers. Finally it was identified that the framework that we proposed was quite effective in evaluating the usability of security mechanisms. However, it is not without its weak spots and does have space for improvement.

6.1 Future Work

There are a number of ways that this study can be built upon in order to improve the research being done around the usability of mobile security mechanisms. The first and likely most obvious is another iteration of this work. If one is to carry out another

iteration of this paper it will further bolster the evaluation framework where similar results are found. It will also bring to light potential issues with the framework where there are differences in results. Likewise, the work could be extended to be evaluated through survey to draw more objective results.

Secondly, the framework could be extended in a number of different ways. For example, this paper currently only measures the usability of seven security mechanisms supported by iOS and Android. Another iteration of this work could apply the framework to new security mechanisms,therefore acquiring a more comprehensive evaluation of the comparison between iOS and Android usability. Likewise, one could extend this evaluation to another mobile platform. As of right now the evaluation has only been used to evaluate iOS and Android security mechanisms. But a natural third platform to evaluate is the Windows platform. Such an evaluation would bring to light an interesting third perspective in the current mobile security climate.

Another way that this work could be improved upon would be by recreating the usability evaluation framework. Our implementation was inspired by Steins iteration, but was greatly simplified in an effort to bring a more clear indication of the respective platforms usability to the reader. Conversely, someone may want to extend the framework to be more widespread and evaluate more categories. Such an evaluation would bring to light some interesting and new results.

Finally, iOS and Android are always coming out with new and improved versions. As a result, this study will likely only be viable for a few more months. As identified in the discussion, as platforms release new mechanisms their usability vastly improve. Because of this, it would be interesting to compare future usability of the same mechanisms with the scores in this study. Such a study would yield informative results in terms of which platform is most actively trying to improve the usability of their security mechanisms.

Chapter 7

DEFINITIONS

AES - Advanced Encryption Standard

API - Application Programming Interface

ATS - App Transport Security

AWS - Amazon Web Services

BSI - Federal Office for Information Security

CA - Certificate Authority

CAVP - Cryptographic Algorithm Validation Program

CBC - Cipher Block Chaining

DSA - Digital Signature Algorithm

ECB - Electronic Code Book

FBE - File-based Encryption

FDE - Full Disk Encryption

GCM - Galois/Counter Mode

HMAC - Hash-based Message Authentication Code

HTTPS - Hypertext Transfer Protocol Secure

IDE - Integrated Development Environment

IV - Initialization Vector

JCA - Java Cryptography Architecture

MACs - Message Authentication Codes

MITM - Man-in-the-Middle

NIST - National Institute of Standards and Technology

OAEP - Optimal Asymmetric Encryption Padding

OS - Operating System

OWASP - Open Web Application Security Project

PRNG - Pseudorandom Number Generator

PKI - Public Key Infrastructure

RCE - Remote Code Execution

RNG - Random Number Generation

SHA - Secure Hash Algorithm

SSH - Secure Shell

SSL - Secure Sockets Layer

TLS - Transport Layer Security

UI - User Interface

UID - Unique Identifier

BIBLIOGRAPHY

[1] Statista, "U.s. data breaches and exposed records 2017 — statistic." [Online]. Available: https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/

[2] "Mobile fact sheet," Feb 2018. [Online]. Available: http://www.pewinternet.org/fact-sheet/mobile/

[3] B. of Govrenors of the Federal Reserve System, "Consumers and mobile financial services 2016," *Consumers and Mobile Financial Services 2016*, Mar 2016.

[4] "Main page." [Online]. Available: https://www.owasp.org/index.php/Main_Page

[5] "Owasp mobile security project." [Online]. Available: https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#Top_Ten_Mobile_Risks

[6] F. Stein, "A framework for the evaluation of mobile platform support for implementing security mechanisms," 2017.

[7] Statista, "Us mobile smartphone os market share 2012-2018 — statistic." [Online]. Available: https://www.statista.com/statistics/266572/market-share-held-by-smartphone-platforms-in-the-united-states/

[8] R. Dillet, "Most ios devices now run ios 11," Nov 2017. [Online]. Available: https://beta.techcrunch.com/2017/11/08/most-ios-devices-now-run-ios-11/

[9] "Owasp mobile security project." [Online]. Available: https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#Top_Ten_Mobile_Risks

[10] "Mobile top 10 2016-m1-improper platform usage." [Online]. Available: https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage

[11] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *J. Vis. Lang. Comput.*, vol. 7, pp. 131–174, 1996.

[12] "What is the software development life cycle (sdlc)? - definition from techopedia." [Online]. Available: https://www.techopedia.com/definition/22193/software-development-life-cycle-sdlc

[13] "Main page." [Online]. Available: https://www.owasp.org/index.php/Main_Page

[14] "Cipher," Jun 2018. [Online]. Available: https://en.wikipedia.org/wiki/Cipher

[15] J. Long, "The evolution of ios security and privacy features," May 2016. [Online]. Available: https://www.intego.com/mac-security-blog/the-evolution-of-ios-security-and-privacy-features/

[16] InspiringApps, "Mobile device security: Data protection on ios and android," Apr 2017. [Online]. Available: http://blog.inspiringapps.com/mobile-industry/mobile-device-security-ios-and-android/

[17] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516693

[18] "hashes digital sigs." [Online]. Available: https://www.davidlprowse.com/article-mac-hashes-digsig.php

[19] "Mobile top 10 2016-m3-insecure communication." [Online]. Available: https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication

[20] [Online]. Available: https://www.feistyduck.com/ssl-tls-and-pki-history/

[21] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516655

[22] "Apple developer documentation." [Online]. Available: https://developer.apple.com/documentation/

[23] "Android oreo — android developers." [Online]. Available: https://developer.android.com/about/versions/oreo/index.html

[24] "Documentation archive." [Online]. Available: https://developer.apple.com/library/content/navigation/

[25] "Security." [Online]. Available: https://developer.apple.com/documentation/security

[26] "Security for android developers — android developers." [Online]. Available: https://developer.android.com/topic/security/index.html

[27] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM,

2006, pp. 506–507. [Online]. Available:
http://doi.acm.org/10.1145/1176617.1176622

[28] "Deprecation," Jun 2018. [Online]. Available:
https://en.wikipedia.org/wiki/Deprecation

[29] "Cryptocompatibility," Nov 2016. [Online]. Available:
https://developer.apple.com/library/content/samplecode/
CryptoCompatibility/Introduction/Intro.html

[30] A. Inc, "Source browser." [Online]. Available:
https://opensource.apple.com/source/CommonCrypto/

[31] "Java cryptography architecture (jca) reference guide." [Online]. Available:
https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/
CryptoSpec.html

[32] "Keygenparameterspec — android developers." [Online]. Available:
https://developer.android.com/reference/android/security/keystore/
KeyGenParameterSpec.html

[33] "Mobile top 10 2016-m2-insecure data storage." [Online]. Available: https:
//www.owasp.org/index.php/Mobile_Top_10_2016-M2-Insecure_Data_Storage

[34] "Cipher — android developers." [Online]. Available:
https://developer.android.com/reference/javax/crypto/Cipher

[35] C. S. Division, I. T. Laboratory, N. I. of Standards, Technology, and
D. of Commerce, "Cryptographic algorithm validation program — csrc."
[Online]. Available:
https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program

[36] "Instruments user guide: About instruments," Sep 2016. [Online]. Available: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html

[37] Google, "google/nogotofail." [Online]. Available: https://github.com/google/nogotofail

[38] "Security certificate errors." [Online]. Available: https://www.digicert.com/ssl-support/certificate-not-trusted-error.htm

[39] "Localauthentication." [Online]. Available: https://developer.apple.com/documentation/localauthentication

[40] Googlesamples, "googlesamples/android-fingerprintdialog," Feb 2018. [Online]. Available: https://github.com/googlesamples/android-FingerprintDialog

[41] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516655

[42] K. Finley, "Half the web is now encrypted. that makes everyone safer," Jun 2017. [Online]. Available: https://www.wired.com/2017/01/half-web-now-encrypted-makes-everyone-safer/

[43] "Network security configuration — android developers." [Online]. Available: https://developer.android.com/training/articles/security-config.html

[44] G. Fleishman, "The huge web security loophole that most people don't know about, and how it's being fixed," Feb 2015. [Online]. Available:

https://www.fastcompany.com/3042030/the-huge-web-security-loophole-that-most-people-dont-k

[45] "Java platform se 7." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/

[46] "Encryption," Jun 2018. [Online]. Available: https://en.wikipedia.org/wiki/Encryption

APPENDICES

Appendix A

EVALUATION WORKSHEET

Evaluation Worksheet

This worksheet should be used as a guide to replicate the evaluation carried out in David Maulick's thesis: "A comparison of the Usability of Security mechanisms provided by iOS and Android.

There are three parts of the evaluation and each contain one or more criteria. The following worksheet will outline step by step how to evaluate each criteria. It will also serve as a place to record the scores that were evaluated and finally create the summary of results figure that will be comparable to the summary results provided by D. Maulick.

This worksheet will serve very useful for anyone that would like to carry out another iteration of the usability evaluation carried out in D. Maulick's thesis. It also would serve to be very effective in a survey format if someone would like to evaluate the usability of these security mechanisms in such a manner.

The motivation for each section in this worksheet is discussed in the actual thesis. Therefore while this worksheet is being filled in please consult the thesis for full understanding of the steps.

Part 1 - Documentation

Coverage

1. Identify pages that you, the evaluator, deem to represent each of the page descriptions of each in the document set for both iOS and Android. Specify a link for each page on the blank lines in figure 1. We suggest using an electronic page in order to simply copy and paste the URLs.
2. For each page in the document set, if the page exists for iOS put a check mark in the iOS checkbox. If the page exists for Android, put a check mark in the respective Android check box.

Pages in Document Set	iOS	Android
Vulnerability home page: iOS: _____ Android: _____	☐	☐
Mechanism home page iOS: _____ Android: _____	☐	☐
Solution iOS: _____ Android: _____	☐	☐
Library docs iOS: _____ Android: _____	☐	☐

Figure 1: Coverage Evaluation Table

3. With the figure 1 table completed.
    a. If any of the functions in the respective platform provided solutions are not documented, fill in the blank for the respective platform below to be 0.
    b. Carry out this step if the platform did not receive a zero on its coverage as a result of missing the library docs. Tally up the number of checkmarks for both iOS and Android's first three documents (The vulnerability homepage, Mechanism home page, and Solution). Add one point for each of those three documents that exist. If all three of those documents exist the platform will receive a 3/3. If only one exists it will receive a 1/3, if two exist it receives 2/3, 0/3 if non of the three exist.

        iOS Coverage: __ / 3
        Android Coverage: __ / 3

Structure

1. Confirm you have identified all documents of document set. The full document set can be found in the thesis.
2. Note figure 2 and 3 below. Figure 2 identifies the structure of iOS documentation. Figure 3 identifies the structure of Android documentation.
3. Note how each page of the document set is represented in figure 2 and 3:
    a. P_Home = Programming homepage
    b. Sec_Home = Security Homepage
    c. Vulnerability_Home = Vulnerability Homepage
    d. Mechanism_Home = Security Mechanism Homepage
    e. Solution = Mechanism Solution page
    f. Library_Docs = library documentation for solution
4. For the respective platforms fill in the respective structure figure; figure 2 for iOS, figure 3 for Android.
    a. For each document in the respective platform's document set:
        i. If link takes you to another page in the document set, draw an arrow to that other page in the respective platform's structure figure
        ii. If link does not jump to another page in the document set; do nothing.

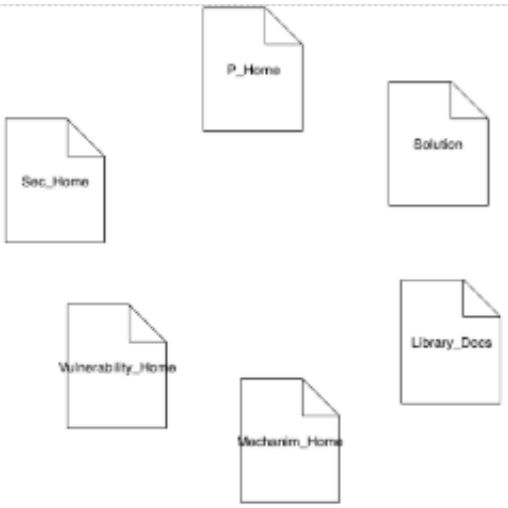


Figure 2: iOS Structure

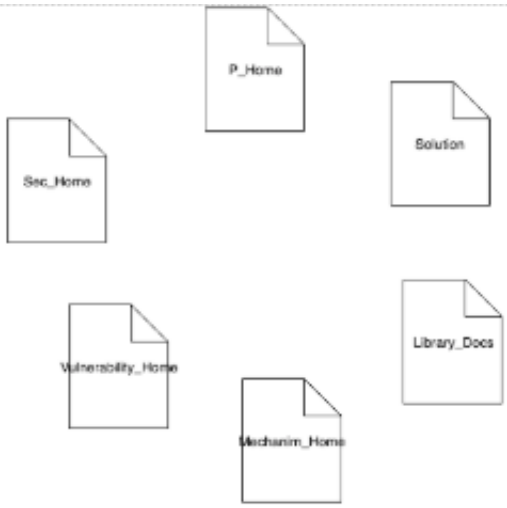


Figure 3: Android Structure

5. Fill in Structure scores in the blanks below for both platforms
    a. If any document in the document set is not connected to any other document in the document set provide a score of 1/2.
    b. If two or more documents are fully disconnected from the other documents in the set, provide a score of 0/2.
    c. If the document set is split into two sub graphs of 3, provide a score of 1/2

iOS Structure: __ / 2
Android Structure: __ / 2

Solution

1. Identify the solution found in the coverage evaluation.
2. Investigate the solution and verify that the solution satisfies the qualities identified in figure 4.
    a. For each quality that it satisfies, put a checkmark in the checkbox next to the quality in figure 4.

Solution Qualities	Satisfied?
Pictured, secure code on a single page	☐
Complete and clear instruction	☐
The solution is up-to-date	☐

Figure 4: Solution Quality Evaluation

3. Tally up the checkmarks and fill in the scores below. Each check mark is one point. A max score is 3/3.

iOS Solution:	__ / 3
Android Solution:	__ / 3

Part 2 - Implementation

For this section you will need to have the solution you plan to implement readily available.

Abstraction

1. Identify all configuration decisions/ choices that are required to implement the security mechanism.
    a. A configuration can be anything from the choice of arguments for a function call, to writing the logic of handling a case.
2. For each configuration decision /choice write a short description of it in the section below.

iOS:

Description 1:

_____

_____

Description 2:

_____

_____

Android:

Description 1:

_____

_____

Description 2:

_____

_____

3. For each configuration decision identify how much the decision relates to the overall goal of the implementation. Then qualitatively identify a final score below that represents the amount of work required to configure the security mechanism. Reference thesis, section 3, for more in depth information on how scoring should be qualitatively evaluated.

iOS Abstraction:	__ / 3
Android Abstraction:	__ / 3

Minimality

1. Identify all cases of overhead work that is constant with the description of overhead work described in section 3 of the thesis.
2. Describe each case of overhead work below for each platform.

iOS:

Description 1:

_____

_____

Description 2:

_____

_____

Android:

Description 1:

_____

_____

Description 2:

_____

_____

3. Identify whether the amount of overhead work identified above outweighs the amount of work associated with the security mechanisms configuration work in the Abstraction evaluation. If it outweighs configuration work, minimality receives a zero. If it is close; 1/2, and if it is minimal; 2/2. Reference thesis section 3 for more information.

iOS Minimality:	__ / 2
Android Minimality:	__ / 2

Robustness

1. Identify the two most likely mistakes that could occur in implementing the respective security mechanism.
2. Describe the two possible mistakes below:

iOS:

Description 1:

_____

_____

Description 2:

_____

_____

Android:

Description 1:

_____

_____

Description 2:

_____

_____

3. For each possible mistake consult the below figure 5 robustness matrix.
   a. Identify whether mistake will hurt the underlying security of the app, or if it will affect solely the security of the mechanism being implemented.
   b. Identify the likelihood of this mistake occuring.
   c. Identify the cross section value in the robustness matrix and record that value below

iOS Mistake 1: ___ / 3 Android Mistake 1: ___ / 3

iOS Mistake 2: ___ / 3 Android Mistake 2: ___ / 3

	Mechanism Security	Underlying Security
Certain	0/3	0/3
Plausible	1/3	0/3
Unlikely	2/3	1/3
Rare	3/3	2/3

Figure 5: Robustness Matrix

4. Identify a final score for iOS and Android robustness below. This score is the lowest of the two scores.

iOS Robustness: __ / 3

Android Robustness: __ / 3

Part 3 - Testing

Addressed

1. Identify if any of the documents in the document set discuss any testing tools for the security mechanism of interest.
2. If a testing tool is identified, verify that the testing tool sufficiently tests the security mechanism.
3. Write the final score below. The score gets one point for a testing tool being mentioned in the document set. The score receives its second point if the testing tool sufficiently tests the implementation. Reference section 3 of the thesis for further discussion.

iOS addressed:	__ / 3
Android addressed:	__ / 3

Part 4 - Fill in summary of Results

After filling out the above worksheet for every security mechanism being evaluated. Fill in the below summary of results table. For each security mechanism that you evaluated the usability of, you should have one worksheet filled out. In order to get you results into a more easily comparable format simply put scores from previous iterations on your worksheets into the correct cross sections of the below table.