

Jali Juhola

SECURITY TESTING PROCESS FOR REACT NATIVE APPLICATIONS

ABSTRACT

Jali Juhola: Security Testing Process for React Native Applications
M.Sc. Thesis
Tampere University
Master's Degree Programme in Software Development
Mar 2022

Nowadays many security-sensitive mobile applications do not create separate applications for every targeted native environment but will use a hybrid mobile framework like React Native as an alternative. These frameworks are nowadays used as alternatives for pure native applications, created separately for every native environment. The problem with these hybrid frameworks is that they create different unique environments, which will make new unique challenges for security validating and testing applications built with hybrid frameworks.

This thesis is limited to only one hybrid framework, React Native, and during the study, a security testing model is created for React Natives security testing purposes. Similar studies have not been previously conducted by using React Native or any other hybrid frameworks using platform-specific native components similarly as React Native uses. Therefore, research was started by defining parts that React Native applications are built with. Relevant parts of React Native for security testing purposes are its three environments. These environments are platform-specific Android and iOS environments, platform-agnostic JavaScript environment and the bridge used to communicate between native and platform-agnostic environments. The model created during the study has the goal of finding vulnerabilities from all of these three environments. This created model improves the current stage of testing React Native applications as the current model commonly used with React Native applications is created for testing only native environments of React Native applications.

At the end, this model is verified during the case study section by conducting the security testing process to a mobile application built by using React Native and by using the created model. Security testing was conducted by using two different groups of tools and methods. These groups of tools and methods are used either with pure native or JavaScript applications.

As a result of the study, it was found that, React Native ecosystem has platform specifics inside its platform-agnostic JavaScript parts. These specifics should be taken into consideration during the security testing process. This also applies to other native component-based hybrid frameworks, where also to gain sufficient security testing coverage, their respective platform specifics should be taken into consideration.

Key words and terms: React Native, Security Testing, OWASP, Mobile, Information Security, Hybrid applications.

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

LIST OF TERMS AND ABBREVIATIONS

IDE	Integrated development environment
XCode	IDE for developing native iOS applications.
Android Studio	IDE for developing native Android applications.
MSTG	Mobile Security Testing Guide
MASVS	Mobile Application Security Verification Standard
Threat	Threat is event or damage which cyber-attack causes to asset or organization. Some common threats could be the disclosure of sensitive data and or assets becoming inaccessible to end customers. [Helfrich, 2019]
Vulnerability	Vulnerability refers to some specific technical security issue in the system. Vulnerability allows an attacker to gain access to the system and enables threat to actualize. Vulnerability could be for example, insecure authentication method used in a development environment that is unintentionally exposed to the production environment. [Helfrich, 2019]
Exploit	Exploit is a piece of code or application that takes advantage of applications or systems vulnerability. [Helfrich, 2019]
Bug	Bug is an error or mistake made by a developer. Bugs are commonly allowing an attacker to create the exploit by taking advantage of the vulnerability.
Black Hat	Black hat is a person attempting to break the system's security without legal permission [Helfrich, 2019].
Risk	Risk in this study is defined as threat and vulnerability paired together. If a vulnerability can be found in an asset and is exploitable, it will allow a threat to be actualized in the asset. These combined creates a risk to that asset. [Helfrich, 2019]

Contents

1. Introduction	1
2. Related work	4
2.1. Mobile Security testing by OWASP	4
2.2. Security testing hybrid applications	6
2.3. Mobile security testing methodologies and models	8
3. React Native	10
3.1. JavaScript	11
3.2. React.js	11
3.3. React.js vs React Native	13
3.4. Architecture	13
3.4.1 JavaScript environment and JavaScript thread	14
3.4.2 Native environments and thread	15
3.4.3 Bridge	15
3.5. Beyond mobile	16
3.6. Expo	17
4. Risk in React Native	18
4.1. Leaky abstractions	18
4.2. Bad JavaScript code quality	19
4.3. JavaScript code obfuscation	20
4.4. Using components with known vulnerabilities	20
4.4.1 Scanning components with known vulnerabilities	21
5 OWASP mobile Risks.....	22
5.1 Improper Platform Usage	23
5.1.1 Testing React Native improper platform usage	24
5.2 Insecure Data Storage	24
5.2.1 Testing React Native insecure data storage	25
5.3 Insecure Communication	26
5.3.1 Testing React Native insecure data storage	26
5.4 Insecure Authentication	27
5.4.1 Testing for insecure authentication	27
5.5 Insufficient Cryptography	28

5.5.1	Testing for insecure cryptography	29
5.6	Insecure Authorization	29
5.6.1	Scanning insecure authorization	29
5.7	Poor Code Quality	30
5.7.1	Finding Code Quality issues	30
5.7.1.1	JavaScript	30
5.7.1.2	Native environments	31
5.8	Code Tampering	31
5.8.1	Testing for Code tampering	32
5.9	Reverse Engineering	32
5.9.1	Testing for reverse engineering	33
5.10	Extraneous functionality	33
5.10.1	Testing for extraneous functionality	34
6	Mobile security testing tools	35
6.1	MobSF	35
6.2	Burp Suite	36
6.3	SonarQube	37
6.4	ApkTool	37
6.5	GitHub Security Advisories	38
6.6	Yarn Audit	39
6.7	Frida	39
6.8	QARK	40
6.9	Insiders	40
7	Vulnerability categorization	42
7.1	Common Vulnerability Scoring System	42
7.2	National vulnerability database	43
7.3	Common Vulnerabilities and exposures	44
8	React Native security testing process.....	45
8.1	Vulnerability assessment	47
8.1.1	Preparation phase	48
8.1.2	Information gathering	49
8.1.3	Scanning the application	50

8.1.4	Result analysis	51
8.2	Penetration testing	52
8.2.1	Exploitations	52
8.2.2	Result analysis	53
8.3	Reporting	54
8.4	Different platforms and model	54
9	Case study security testing React Native application.....	55
9.1	Preparation phase	55
9.2	Information gathering	56
9.2.1	Android	57
9.2.2	iOS	58
9.3	Scanning the application	59
9.3.1	JavaScript environment	60
9.3.2	Android	63
9.3.3	iOS	65
9.4	Result analysis	66
9.4.1	JavaScript	67
9.4.2	Android	68
9.4.3	iOS	69
9.5	Exploitations	69
9.5.1	JavaScript	70
9.5.1.1	Preparing the attacks	70
9.5.1.2	Exploiting the vulnerabilities	71
9.5.2	Android	72
9.5.2.1	Preparing the attacks	72
9.5.2.2	Exploiting the vulnerabilities	73
9.5.3	iOS	73
9.6	Result Analysis	74
10	Conclusions	76
10.2	Limitations	77
10.3	Future work	77
	References	82

1. Introduction

The environment where multiple mobile operating systems should be taken into consideration can be challenging when developing applications. Nowadays, multiple platforms should be targeted to gain sufficient coverage among different devices. In addition, code cannot be shared between these targeted platforms as those are implemented by using different technologies. Therefore, nowadays, different teams are required to create applications for each targeted platform. At the time of writing, most companies will target only two platforms Android and iOS, which are holding the majority of the market with a combined market share of 99 percent [Statista, 2021].

The solution has been introduced to mitigate the issue of needing to create multiple seemingly similar applications with distinct codebases that are used only to target different platforms. This solution is hybrid frameworks like React Native, Flutter or Ionic. These frameworks allow the creation of native-looking applications, which are executable in multiple platforms simultaneously by using only a single codebase. Although these frameworks are saving time during the upkeep and development phases, the question is raised about how secure are these hybrid frameworks? When an application built with a hybrid framework is compared to a pure native application, it becomes apparent that the attack surface of hybrid applications is more extensive than natively built applications one. That is because hybrid application includes in addition of attack-vectors of native applications, the hybrid framework as itself. The hybrid framework is working as a platform-agnostic part of the hybrid applications and will create another attack surface for the application. That attack surface in React Native, as an example, includes the whole JavaScript engine, which in Android's case is packaged separately inside the application package [Mueller, *et al*, 2020].

The aim of this study is to create a comprehensive security testing model with a suitable toolset for security testing React Native applications. According to Mueller and others [2020] and Hale and Hansson [2015], the security testing process of React Native applications includes two parts. These are the native parts of the applications and hybrid framework-specific parts of the application. To test native parts of the applications ten most common mobile risks by OWASP [2016] are used as a base. Each of these ten risks are studied separately to determine the methodology and toolset for testing these risks. There has not been yet a similar risk listing created for React Native-built applications. Therefore, documentation and online sources like Facebook [2019] and CossacLabs [2021], are used to create a React Native specific risk listing, which will be studied similarly as Owasp Mobile Top Ten [2016], to create a methodology and toolset for these React Native particular risks. At the end of the study, this model and toolset are validated

by testing real-world React Native application and documenting findings and issues encountered.

This study is divided into ten chapters. Starting with the related work section where a literature review about security testing native and hybrid mobile applications is conducted. In addition to these generic security testing searches, tools and methodologies for testing mobile applications will be studied with the literature search later in that chapter. Related work found in this stage works as background and is used in chapters where React Native, native mobile applications, and security issues related to React Native and native mobile applications are studied. After, tools for security testing these previously mentioned issues and methodologies for categorizing vulnerabilities found are studied. By using these previously described pieces of information, a comprehensive security testing model for testing React Native applications is created. At the end of the study, this model is validated by conducting security testing to production-level React Native application consisting of sensitive data.

There are different terms used to describe the security testing process of mobile applications. These are mobile application security review, mobile application security testing, and mobile application penetration testing. These all are terms referencing according to Mueller and others [2020] the same process. This security testing process can be divided at the higher level into three categories automatic, semi-automatic, and manual testing. Usually, automatic analysis parts of the testing process are executed in the vulnerability assessment phase. In the vulnerability assessment phase, source code and a package of application are reviewed for vulnerabilities and risky behaviors by using automated tools like MobSF and Drozer, which are introduced later in the thesis. Problem with these automatic scanning tools is that they are lacking the sensitivity of testing more subtle errors found in business logic and organization-specific assets [Boduch, et al. 2020]. Therefore, in the vulnerability assessment phase, parts of the application might be beneficial also to review manually. This would include for example, a possible custom authentication scheme used by the application [Mueller, *et al*, 2020]. Therefore due to these manual steps, the vulnerability assessment process as a whole is usually a semi-automatic process.

After the vulnerability assessment phase, a set of unvalidated vulnerabilities is gathered. In this stage of the testing process, there is a high possibility that those unvalidated vulnerabilities are false positive. This is why in the next phase, a tester will validate this set of vulnerabilities by actively attacking the application like a real attacker would. This makes it possible for security tester to gather valuable information about the damage that these vulnerabilities would cause and the difficulty of exploiting these vulnerabilities. This exploitation phase of testing can be a time-consuming and costly process to execute, requiring a lot of time from a highly skilled tester with knowledge

about technologies used by the target system. On the other hand, vulnerability assessment is at least partly automatically runnable, making it more easily executable as often as needed [Umro, *et al* 2012]. This is why it is common that the penetration testing phases of security testing processes are being neglected and only the vulnerability assessment phase will be executed when application is tested [Mueller, *et al*, 2020]. However, according to PTES [2021] carefully executed penetration testing is a crucial part of the successful security testing process and if executed correctly, it will supply critical information about the system's overall state of security.

In addition to the automation level of testing, the type of testing process also depends on data that the tester is given about the target system. This data given is commonly divided into three different groups which are:

Black box testing is the way of the testing system like a real malicious user would by downloading the application from its official source and conducting attacks by using only publicly available information about the application. In the context of mobile applications, full black box testing would mean that application is tested obfuscation activated without the source code.

Gray box testing Is a way to conduct testing with limited information about the application. This would mean in the example, the application being un-obfuscated or a tester having valid admin login credentials.

White box testing in many sources this process is referred also as internal security auditioning. In this process, a tester has full access to the architecture and source code of the application.

Black box testing resembles the situation with an actual external attacker and therefore is the most realistic situation. On the other hand, black-box testing makes it difficult for testers to verify possibly unintended behaviors and conduct penetration testing [Mueller, *et al*, 2020]. Second problem with black box testing comes with the difficulty of running time-saving automated security tools. Many of these tools will require either unobfuscated binaries or source code as an input [MobSF, 2021; SonarQube, 2021]. So overall black box testing can be challenging in situations where testers have strict time constraints for the testing process. Therefore it is highly advisable for testers, to ask the application to be de-obfuscated and a source code revealed during the testing. This is because obfuscation of mobile applications by itself is not effective security control [OWASP, 2016]. Access to source code and architecture can also be important in situations where applications have previously not gone through security testing processes before. This makes an initial testing process much more efficient and sensitive to more subtle vulnerabilities and risks [Boduch, *et al*. 2020].

2. Related work

In this section, the literature review is conducted using scientific articles searched with Google Scholar and Andor services. Andor is a service owned by Tampere University which consists of access to articles from 420 databases when writing the thesis [Tuni, 2021].

2.1. Mobile Security testing by OWASP

The first search is related to work regarding Mobile Security testing and safe mobile development at the generic level, without any technologies specified. As OWASP mobile top ten risk listing and OWASP methodology are used by this study, this search is limited to OWASP related methodologies. The first search is done by using two different queries first is *OWASP AND Mobile AND (Security testing OR Penetration testing)*, which is used to search OWASP related literature about security testing and penetration testing mobile applications. Second search is done by using the query *OWASP AND Mobile AND Development*. In the literature review, literature about safe development methodologies by using OWASP methodologies is searched.

Literature regarding security testing and secure development of mobile applications by using OWASP methodology was found. These studies are based on two different sets of risks. OWASP published its first set of risks in 2014. Two years later, these risks became somewhat obsolete when OWASP published a newer and currently latest set of risks in 2016. Although the older group of risks can now be regarded as outdated, OWASP included most of the earlier risks in the newer 2016 listing. Therefore, studies conducted with an older 2014 set of risks are at least partially comparable with studies including more recent 2016 risks. [Borja, *et al* 2021]

Rodríguez and others [2018] studied safe development methods of mobile applications by using OWASP methodologies as a guideline by using a newer set of risks provided by OWASP. In this study, these risks were introduced, and the importance of considering these risks already at the development phase was emphasized. However, the study is focusing more on the development phase of the mobile applications, and therefore, no testing process or methodology is introduced during this study.

There were also studies found where penetration testing, vulnerability assessment, or both are conducted. A couple of examples of studies like that are Alanda and others [2020] and Borja and others [2021]. However, these studies conducted their testing process either to Android or an undisclosed operating system. An unspecified operating system makes it harder to compare these studies to our results at the end of our study. The second problem with these two studies from this thesis perspective is that they do not disclose the tools or testing methodologies used.

Two studies were found, which were conducted by using applications containing medical data—however, an earlier 2014 set of OWASP Mobile Risks were used, which

makes the results of these studies only partially applicable to this thesis. In this study, two Android medical applications were studied by using a full security testing methodology. This methodology included vulnerability assessment and penetration testing phases. Critical issues regarding sensitive data being leaked through unsafe storage were found from both target applications. This study introduced partially how these issues were found by including information about the methodology used. This study had a similar summary as the first study, which states that OWASP risks should be used already during the development phase of the applications, and development teams and security testers should be better educated about these OWASP risks and their prevention methods [Acharay, *et al*, 2015].

A second study is using an older 2014 set of risks to test Android medical applications. This study was conducted by Cifuentes and others [2015] and in this study, 60 medical applications, including medical data, were analyzed. Analysis was done by using a commercial IBM vulnerability assessment tool. The composed results of these 60 applications, which were tested can be seen in Figure 1. Results of the study were different than in the study conducted by Acharya and others [2015]. In this study, most of the found vulnerabilities were categorized as untrusted inputs, which are not prioritized as a common risk according to OWASP [2016]. Static analysis tools in the web and mobile environments are known to report false positives when scanning untrusted inputs and client-side injections [Wang and Alshboul, 2015]. These false positives can be avoided by conducting manual penetration testing or validating found vulnerabilities by other means. In the study by Cifuentes and others [2015], there was no mention of manual validation of these vulnerabilities and the IBM tool used to carry out vulnerability assessment is not free-to-use and therefore not accessible for this study.

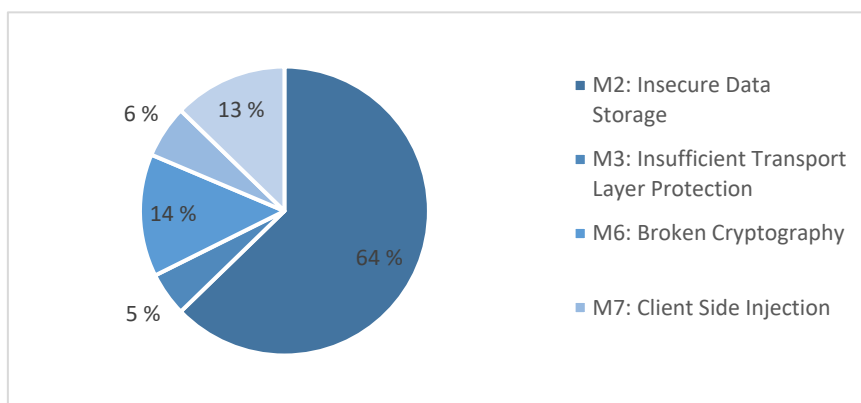


Figure 1 Results of found vulnerabilities categorized to OWASP risk listing [Cifuentes, *et al*, 2015].

In a study by Bojjagani and others [2017], Five Android and three iOS mobile banking applications were tested using vulnerability assessment and these results were validated by penetration testing. Contrary to studies introduced earlier, this study provided a full

description of a toolset and methodology created in the study and used during the testing process. Testing methodology of the study was overall very similar as provided by OWASP [2020].

2.2. Security testing hybrid applications

Literature review conducted with the first set of keywords, all literature found was about pure native applications mainly targeting the Android platform. In addition to the lack of iOS-related literature, there are no studies about the security of hybrid applications found. That is why new search by using a second set of keywords is conducted. In this search, security testing and secure development are researched in situations where hybrid frameworks like React Native or Flutter are used. Queries used to conduct this search are *OWASP AND Mobile AND Hybrid, Security AND Mobile AND Hybrid* and *Hybrid SECURITY TESTING AND (React Native OR Flutter OR Ionic)*.

Borja and others [2021] conducted a more generic Android Application security analysis using the newer 2016 OWASP Top Ten Mobile Risks. This study differs from other studies. Because it states that applications built with frameworks like React Native and Flutter can be tested similarly like pure native applications without specifying the hybrid framework used [Borja, *et al*, 2021]. This study can give the base for security testing Android applications. However, in this thesis, the aim is to create a complete approach for testing hybrid applications. That means in this study's context that hybrid application specifics are taken into consideration. These known React Native framework specifics stated in documentations are at least JavaScript obfuscation, network security, and storing the sensitive information that should be considered when testing React Native applications. [Facebook, 2019; OWASP, 2021]

A thesis conducted by Wällstedts [2019] studied the security of JavaScript-based hybrid applications. However, more restricted React Native's version Expo was used. In Expo developer of the application is prohibited from writing custom native code, making the native environment entirely inaccessible to developers [Expo, 2021]. This study conducted security verification of real-world application by using Expo and a generic OWASP Mobile Application Security Verification Standards (MASVS) security checklist. Illustration of the checklist used by Wällstedts can be seen in Figure 2. Currently, this generic OWASP MASVS is outdated for security verification purposes of React Native applications. This is due to that, in August 2019, OWASP released React Native customized MASVS [2020]. So currently, at the time of writing this, it is advisable to use this newly created document during security verification of React Native applications.

#	Description	Expo	RN
2.1	System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.	✓	✓
2.2	No sensitive data should be stored outside of the app container or system credential storage facilities.	✓	✓
2.3	No sensitive data is written to application logs.	✓	✓
2.4	No sensitive data is shared with third parties unless it is a necessary part of the architecture.	✓	✓
2.5	The keyboard cache is disabled on text inputs that process sensitive data.	✓	✓
2.6	No sensitive data is exposed via IPC mechanisms.		
2.7	No sensitive data, such as passwords or pins, is exposed through the user interface.	✓	✓

Figure 2 OWASP MASVS Security checklist part about data storage and privacy [Wällstedt, 2019]

In a study conducted by Brucker and Others [2016] statical code analysis tool is created for a hybrid framework Cordova. Before implementing the tool, Cordova's attack surface was determined to contain three components, web-environment-related issues like XSS, common mobile-environment-related issues like privacy leaks, and cross-language calls [Brucker, *et al*, 2016]. Study conducted by Hale and Hansson [2015] on the other hand created a process for analyzing vulnerabilities in hybrid applications created with frameworks like Apache Cordova. This process is based on five steps, which are (1) resource landscape, (2) Vulnerability assessment, (3) Creating attack vectors (4) Exploiting attack vectors, and (5) Assessing the results.

At the time of writing there is a lack of study in the area of React Native security testing or any application, which is developed using a hybrid native interface-based framework like React Native. According to Mueller and others [2020], testing hybrid applications is a generally similar process to testing purely native mobile applications, which is the case with React Native where all user interface components are the same native ones used during pure native development. Therefore, MSTG methodologies can be used when testing native parts of these hybrid applications. However, some differences can also be found. One significant difference between React Native and native applications is the JavaScript engine responsible for executing applications logic. This engine is embedded inside React Native applications [React Native Documentation, 2021]. In addition to the JavaScript engine, React Native also has other unique risks and vulnerabilities, some of which are defined in React Native Security Documentation [2019] and React Native customized MASVS documentation [2019]. So, in this study suitable penetration testing and vulnerability assessment models, including tools, are examined. The goal of the study is to answer the question: Are only OWASP MSTG methodologies enough to test React Native applications, or are there additional threats that should be considered?

2.3. Mobile security testing methodologies and models

In this chapter literature review with the third set of keywords is conducted in the area of Mobile security testing methodologies and models with and without OWASP methodologies and risks. The goal of this section is to find a base for the model that will be used in the case study at the end, and for that reason, it is essential that the model selected is compatible with OWASP testing methodologies and risks. This search was conducted by using query *(OWASP OR Mobile) AND Security AND (MODEL OR METHODOLOGY)*.

Mobile security testing can be challenging due to the large variety of devices, uniqueness of these environments, and lack of global standards. Wang and Alshboul [2015] proposed four different sets of methodologies for validating and testing the security of mobile applications. These methodologies are mobile forensics, penetration testing, static and dynamic analyses [Wang and Alshboul, 2015]. These methodologies can be grouped further by combining mobile forensics with static and dynamic analysis. Mobile forensics is excluded since it refers more to recovering mobile data from operating systems or devices level. Protection of application data is studied more accurately when application storage solutions are studied. Secondly, static and dynamic analysis can be grouped under the same generally used term, vulnerability assessment [Mueller, *et al* 2020]. Vulnerability assessment is generally used as a term that refers to gathering vulnerabilities from the application using automated or semi-automated tools [Haq and Khan, 2021]. This study uses these two previously defined security testing methods to create a model, which is applied at the end to React Native application. This study uses OWASP Mobile Top Ten risks as a core for determining risky areas of applications security. These ten risks can be tested with the methodology proposed by Palacios and others [2019], which is OWASP MSTG. MSTG is used as a reference and step-by-step guide to security testing native iOS and Android applications.

		Pen-Test Methodologies and frameworks				
		OSS TMM 3	NIST-ISAM	OWASP-MASVS	ISSAF	PTES
ISO/IEC 25010:2013 software quality model	Coverage	A	PA	A	PA	PA
	Flexibility	PA	A	PA	PA	PA
	Adoptability	PA	PA	A	NA	PA
	Modelling	NA	NA	NA	NA	NA
	Usability for amateur developers	NA	PA	PA	NA	PA

Legend : A: Applicable P: Partially Applicable NA: Not Applicable

Figure 3: Different penetration testing methodologies reviewed [Haq and Khan, 2021]

Haq and Khan [2021] conducted a systematic literature review regarding penetration testing methodologies of mobile and web applications. In this study, five different

penetration and security testing methodologies were selected and reviewed. Results of this systematic literature review of models are displayed in Figure 3. These methods are reviewed more precisely later during the study and one is chosen to create a base for the security testing model.

3. React Native

React Native, according to its documentation, is a hybrid user interface building framework that is created by Facebook [React Native Documentation, 2021]. The framework was initially created to capitalize on Facebook's previously released web user interface building library's success [Facebook B, 2020]. React-Native was initially released to the public in 2015. Before its public release, it worked for a while as Facebook's internal tool for creating hybrid mobile applications by using JavaScript.

There have already been hybrid mobile application building tools before the release of React Native. These applications were created using HTML5 and WebView based technologies. The idea behind these older frameworks is similar to React Native one, which is that a team without any notable platform-specific knowledge should be able to develop a native application-looking website displayable inside a native application wrapper, which is similar to the React Native WebView component. This application is then distributed through Google Play and Apple App store, like a typical pure native application. A widely known example of WebView based hybrid application framework is Apache Cordova, which also can use device-specific native APIs through plugins [Apache, 2021]. Most notably, from customers' point of view, Cordova applications can look like typical native applications with poorer performance due to an HTML rendering engine directly manipulating DOM.

The important part about frameworks and libraries, when an application with a long life cycle is created is long-term maintenance and active ownership of the framework or library. A second important part is the overall trust and security of the framework. That can be determined by examining the maintainers and current usage level of the library. It can be determined whether the library is likely to have maintenance and has active bug fixes far into the future, which should not be taken for granted as it is a problem in the open-source world [Zimmermann, et al 2019]. Library or framework being unmaintained can lead to even major security issues with bugs not worked on early enough stages or never [Decan, et al 2018].

React Native is an open-source framework [Facebook A, 2021]. This can lead to security risks as open-source projects are commonly left unmaintained [Zimmermann, et al 2019]. That, however, does not seem to be a likely outcome for React Native as Facebook created a framework with a track record of successful and well-maintained JavaScript open-source projects under its ownership [Facebook E, 2021]. According to its GitHub page React Native also has in addition to direct support from Facebook substantial open-source community maintaining it with a community consisting of over 2000 unique contributors at the time of writing this [Facebook A, 2021]. According to React natives home page, React native is also widely used across the different companies representing different industries, including Facebook, Tesla, Instagram, Oculus,

Salesforce, and Skype [Facebook A, 2020]. So overall, it is likely and can be said relatively confidently that React Native is well maintained and mature technology with security problems being addressed in a timely manner.

3.1. JavaScript

JavaScript was first introduced in 1995 by Netscape as the scripting language for web browsers. The idea behind JavaScript is to make websites interactive without reloading the application from the server. While JavaScript is best known as a scripting language for browsers, nowadays, its use cases have widened all the way to Node.js backends, React Native mobile applications, and the CouchDB database engine. [Mozilla, 2021]

JavaScript can be considered as one of the greatest strengths of React Native when considering JavaScript's popularity. When JavaScript is combined with its type-extended version Typescript, the whole ecosystem consists of little under 40 percent of all pull requests opened on GitHub [GitHub pull request statistic 2020]. This number is by far the largest amount of pull requests by the programming language in GitHub. In contrast, according to GitHub statistics most used native mobile development programming language is Java which is used as a language in the Android environment and has around 9.2 percentages of popularity. Lastly rarest languages used in the mobile world are on native iOS projects. The rarest language in iOS environment would be Swift which is used with approximately 0.6 percentages of pull requests opened in GitHub [GitHub pull request statistic 2020].

3.2. React.js

Before creation of libraries like React, web DOM was usually directly manipulated as shown in Figure 4. Displayed code snippet uses jQuery, which describes how the user interface works while the transition between states is happening. React.js on the other hand has an approach where the developer creates components and components are representing how data is displayed to the final user. When data inside the component is changed runtime React will handle necessarily changes to the user interface. [Facebook B, 2020]

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <script>
5     $(document).ready(function () {
6       $("#hideButton").click(function () {
7         $('#thesis').hide();
8       });
9     });
10  </script>
11 </head>
12 <body>
13   <div>
14     <div id='thesis'>
15       <h1>React Native</h1>
16       <span>Security</span>
17       <span>60</span>
18     </div>
19     <button id='hideButton'>
20       Hide item
21     </button>
22   </div>
23 </body>
```

Figure 4 Example about jQuery code

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 class Thesis extends React.Component {
5   constructor(props) {
6     super(props);
7     this.state = { thesisVisible: true };
8   }
9   componentDidMount() {
10    // actions immediatly after component is inserted to DOM
11  }
12  render() {
13    return (
14      <div>
15        {this.state.thesisVisible && (
16          <h1>{this.props.title}</h1>
17          <span>Subtitle: {this.props.subtitle}</span>
18          <span>Pages: {this.props.pages}</span>
19        )}
20        <button onClick={() => this.setState({thesisVisible: true})}>
21          Hide item
22        </button>
23      </div >
24    )
25  }
26 }
27 ReactDOM.render(
28   <Thesis title='React Native' subtitle='Security' pages={60} />,
29   document.getElementById('root'))
```

Figure 5 example about React code

In the example, Figure 5 describes React's main functionality, which is displayed with the class component. In the declaration of class component, the state is defined in line 7. The state is part of the React component, which has functionality to store data runtime inside the component. The state is initialized before first rendering and is mutable after that with *setState* function [Facebook B, 2021]. When state data has been changed, React will automatically reflect those changes to rendered output. When optimized and created correctly according to documentation, re-render will only affect the parts affected by a change of the state [Facebook B, 2021].

Properties on the other hand, are similar to state values. The only two major differences between state and the properties are that properties are immutable and are passed to the component from outside. [Facebook B, 2021]. Due to these qualities' properties are immutable from the components point of view and can only be edited or replaced from the outside. An example about the properties can be seen in line 28 of Figure 5,

React does not have a lot of different APIs exposed to the developer. The most common and essential functionality React.js offers is displayed in Figure 5. According to Facebook [2020], the simplicity of React will make it easier for developers as time is not spent used to learning continuously new things like in larger frameworks.

3.3. React.js vs React Native

If React.js would have to be explained to a person without prior programming experience or who hasn't heard about it before, I would define React like it was described in React's own website as simply a library for building user interfaces [Facebook F, 2021]. All use cases of React can get confusing when actual Facebook's documentation is used as reference. An example of this is defined in the React.js documentation [2021]. It states that: "Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep the state out of the DOM." Like in many other places in that documentation, Document Object Model (DOM) is referenced in the previous quotation. The problem is that DOM is used as the structure of HTML web pages and is not necessarily known for example, in the mobile world. So, in this study, React.js is used to manipulate web DOM and create web interfaces. On the other hand, if other than web user interfaces are studied, these are called in this study native interfaces and therefore are developed using React Native.

3.4. Architecture

React Native codebase and final application package is composed of two main parts Native code (Java, Swift, and Objective-C) and React.js like JavaScript code. The first problem that Facebook engineers had to face when React.js was ported to different native environments was that when there are multiple diverse native ecosystems there are usually multiple different languages executed in different isolated environments. At the

beginning when development was started, those isolated environments did not have any way of communicate between each other. This is the problem that was solved with React Native's component bridge. The bridge is the pathway of communicating between these different environments. High-level architecture of React Native can be seen in Figure 6. [Facebook F, 2021]

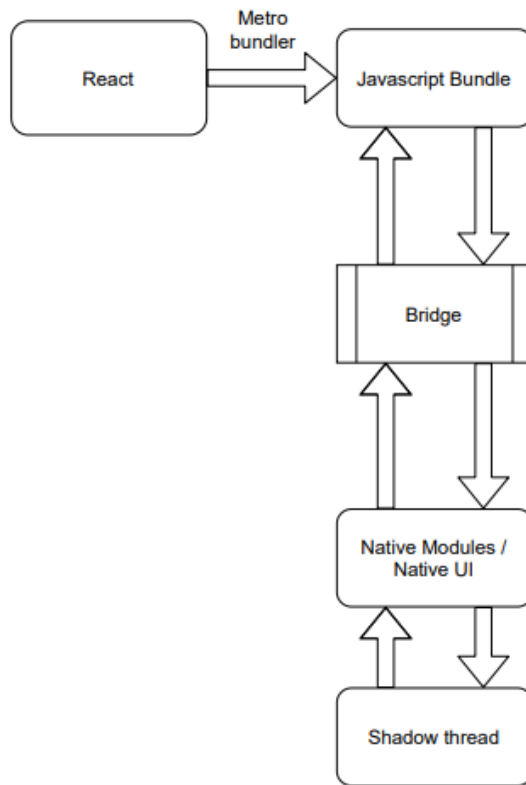


Figure 6 React Native architecture

3.4.1 JavaScript environment and JavaScript thread

At the lowest level, most of the React-Native's applications and all JavaScript environment's code is written in JavaScript. JavaScript environment and code in React Native project works as platform-agnostic parts of React Native application. Most of this JavaScript code is displayed with React syntax, which as itself is not valid JavaScript, interpretable by JavaScript engine and therefore, is not directly installed to the devices [Facebook F, 2021]. In order to React code being executable by JavaScript engine, React code is transformed to valid JavaScript code by using Metro bundler, which has similar functionality as well-known web application alternative Webpack.

In React Native, business logic is defined to a platform-agnostic JavaScript environment. This means in context of the device that business logic of the software is

located at JavaScript bundle [Facebook C, 2021]. This will cause dependency to all installed React Native applications, which will mean that all React Native applications should have the same JavaScript engine to execute the business logic of the application. In the case of React Native, it was decided that that open-source JavaScriptCore is used. This is due to fact that all iOS devices have JavaScriptCore already included inside the Safari browser [Apple, 2021]. However, the engine was not included inside Android operating system, and for that reason, Metro Bundler will bundle JavaScriptCore inside android applications which will make even Hello World applications around 3 to 4 megabytes [Facebook B, 2021].

3.4.2 Native environments and thread

In React Native, the JavaScript environment's code is written in JavaScript and should be platform-agnostic, where a single line of code will be the same in iOS and Android applications. The native environment, on the other hand, is in some sense opposite. In different Native environments, code must be written separately to all different platforms. That will mean using the native languages interpretable by targeted devices native environment. React Natives native environment languages are including Objective C, Swift, Java, and Kotlin. In this study's context, these native environments are Android and iOS. In React Native these different native environments are separated by creating their own platform-specific directory to the root of the project to each targeted native platform [Facebook D, 2021].

Although even if native-specific code is not written nor visible in source code and everything is working with only JavaScript code, it should not be forgotten that native environments for all targeted platforms will still exist as a base in React Native project. That will be evident as soon as the application's start-up, where the opening application will first execute a native environment like in entirely natively written applications [Facebook B, 2021]. Also, in the lower level, everything users are seeing in their mobile devices is still native. That means all view components of the application like React Native's view component which is at the device level either of these native components UIView or Android.view depending on the environment project is executed [Facebook D, 2021]. In that sense, JavaScript is only an abstraction of multiple different native environment-specific properties that are implemented at the lowest level by using the native code [Facebook B, 2021]. Also, in addition to native components, React native uses platform-specific UI events like swipe and press.

3.4.3 Bridge

Now we have two separate and isolated environments wrapped inside a single application. One environment commonly used as running browser used language JavaScript and other

commonly used to run native operating-system-specific mobile code. That is where bridge is used in React Native Architecture of the bridge is illustrated in Figure 7.

React Native Bridge is implemented using C++ and Java and the goal of the bridge is to enable a two-way communication channel between JavaScript and native modules. For example, when the application is launched, the native entry point of the application is the initial point of that launch. This starts the JavaScript environment and sends native application commands to start the actual JavaScript application. This bridge is communicating between environments using JSON serialized data. According to Facebook [2021], this communication channel between environments has goal of making React Native applications faster than Native applications.

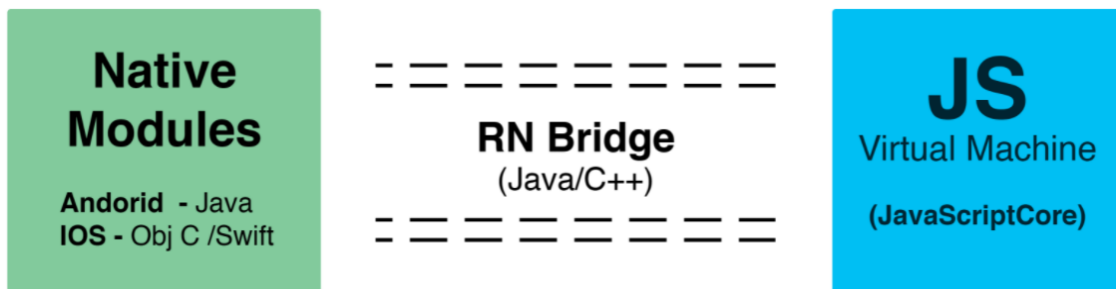


Figure 7: React Native bridge between JavaScript environment and Native environment [Facebook F, 2021].

3.5. Beyond mobile

React Natives official documentation has two extensive platform-specific guides for iOS and Android [Facebook D, 2021]. These same two platforms are also mentioned in most pages when there are discussions about UI layouts. So, is React Native a purely hybrid framework implemented to two native platforms, Android and iOS, or is there a bigger plan for React Native as a whole?

Inside React Native development teams blog posting [2021], React Native's plans for the framework's future and moving beyond mobile have been discussed. It is revealed that React Native is looking for new managers for React Native VR and React Native desktop at the end of the blog post. According to the blogpost, this new direction for React Native would make the framework a truly platform-agnostic solution. The idea is that developers would be writing code for a user interface that React Native would automatically port to all possible platforms from mobile to virtual reality [Facebook C, 2021].

This study is reviewing security of React Native's Android and iOS ecosystems. However, React Native already has community-maintained porting to smart television which itself is not in scope of this study. These already existing community-maintained projects and currently official developed and unpublished ports to React Native are necessary to keep in mind. This is due to a possible whole new direction for React Native

where already existing JavaScript abstraction could possibly reach more different types of platforms with entirely different requirements and widely different native APIs.

3.6. Expo

In the Expo documentation [2021] Expo is defined as a framework and a platform for creating universal React applications using set of tools and services built around React Native [Expo, 2021]. This sounds similar to React Native; however, one major difference is managed workflow like an Expo documentation calls it. In Expos managed workflow, developers are using only JavaScript environment, and handling of native environment is left entirely to framework's responsibility. Expo does this by prohibiting users from downloading custom platform-specific libraries and writing custom platform-specific code or configurations [Expo, 2021]. However, as a trade-off expo gives a large amount of well-tested and self-developed libraries, which are built in top of React Native. This will outsource most of your React Native applications' possible areas of bugs and attack vectors and will significantly diminish the need for the development team's platform-specific knowledge; however, as a trade-off Expo restricts what developers can do with the application.

4. Risk in React Native

In the end, a major part of React Native applications' functionality needed to be security tested is the same as functionality in pure native applications [Mueller, *et al* 2020]. Therefore, the main part of security testing React Native applications is to test applications according to platform-specific testing literature like provided by Mueller and Others [2020]. Thus, the primary research of security and mobile risks is conducted in chapter 5, where OWASP Mobile Top Ten risks for pure native applications are introduced. The goal of this section is to find React Native-specific security issues. These security issues are then combined with MSTG provided pure native security testing methodologies to create a more comprehensive testing methodology for React Native. These risks will be mapped to OWASP Top Ten Risks defined later at the end of chapter 8 to create a singular process and risk listing for testing React Native applications.

The only related study found regarding React Native security testing was a master's thesis written by Wällstedt [2019]. In this study, React Native applications security was verified using the generic OWAPS MASVS listing. However, this is not anymore the best method for testing React Native applications. According to Mueller and others [2020], MASVS methodology is partly applicable but not complete when testing hybrid applications. Therefore in this section, React Native-specific security issues are studied using documentation and React Native-related security-related popular literature. At the end of this chapter, possible testing methods are proposed. Results of this section are described and mapped to OWASP Top Ten Risks in Table 1.

React Native issue	OWASP Top Ten
Leaky Abstraction	Improper Platform Usage
JavaScript Code Quality	Client Code Quality
JavaScript Code Obfuscation	Reverse Engineering
Using Components with Known Vulnerabilities	Client Code Quality

Table 1 Found React Native related issues

4.1. Leaky abstractions

In React Native single JavaScript function implemented at the native level has possibly different implementations in different native platforms. This functionality can be different depending on whether an application is built in a native iOS or Android environment. If these inconsistencies between environments are found, they can have a high impact on the application's security [OWASP, 2016]. A well-known example of this is when

sensitive data is stored on the device using SecureStorage implementation. According to OWASP MASVS [2021], when the application uses an L2 security level, sensitive data should be erased from the device after reinstallation of the application. However, in iOS, data in the keychain will persist over reinstallations, and in Android data will be removed during reinstallations. That makes the functionality of securely storing data in Android and iOS different and causes a leaky abstraction-related risk to iOS applications. A second major difference between platforms is with the same component SecureStorage and its encryption implementations in iOS and Android platforms. In iOS devices, keychain data is decrypted when a device is unlocked, by using a passcode or biometric scanner [Apple Documentation, 2021]. Android on other hand, decrypts data only when data is used by the application. This causes an issue according to security standards of OWASP MASVS [2021], which states that when using the L2 security level, all sensitive data should be encrypted before being placed on the iOS keychain. Different Safe and unsafe storage solutions in native environments are displayed in Table 2 [CossacLabs, 2021].

	iOS (Keychain)	Android (SharedPreferences + KeyStore)
Data stored encrypted	+	+
Data persists across app reinstalls	+	=
Hardware-backed encryption	+	+ / =
Data decrypted only before usage	=	+

Table 2 Differences between operating systems when using React Native SecureStorage [CossacLabs, 2021]

For testing leaky abstraction risk, there are currently, at the time of writing, no tools or solutions available to scan leaky abstraction-related issues either automatically or semi-automatically. Therefore, as stated in many sources, when multiple native functionalities are abstracted to a single function on JavaScript level, a team should have knowledge about functionality implemented in both Android and iOS environments when using React Native [Facebook, 2019].

4.2. Bad JavaScript code quality

When statically assessing pure native applications, code analyzed is either native code of iOS or Android. In React Native applications, all business logic, when built correctly by using React Native Documentation [2021], is written in JavaScript. Therefore when assessing React Native applications, security analysis of JavaScript-specific vulnerabilities like usage of *eval* function should be conducted. There are many tools for

statically analyzing JavaScript code. The most optimal tool would be analyzing a JavaScript code from the perspective of React Native. However, such a tool was not found, and therefore, a more generic tool, SonarQube is selected to conduct a static analysis of JavaScript code. SonarQube is presented more in detail in chapter 6.

4.3. JavaScript code obfuscation

According to OWASP [2021], binary protection is recommended for mobile applications at least in situations when applications are storing or handling sensitive data. In popular literature, there are multiple occasions where JavaScript obfuscation in React Native project has been discussed [Stackoverflow A, 2019; Rguez-Sánchez, 2019]. Third-party tools like JScrambler have been proposed as a solution to obfuscate JavaScript code in React Native projects [JScrambler. 2021]. Zhang and Others [2021] proposed method for obfuscating React Native Android applications by using the tool called ProGuard. ProGuard is included in the Android development kit. ProGuard is commonly used when obfuscating hybrid applications [Zhang, *et al*, 2021]. However, the article did not mention if obfuscating hybrid applications with ProGuard will also obfuscate JavaScript code or does it work only on a native code level.

In the case study section, when binary protection of application is tested, and JavaScript bundle of the project is reviewed. Obfuscation of the JavaScript bundle will also be inspected. However, implementing JavaScript obfuscation might be unnecessary for applications where highly sensitive data is not stored or transmitted or no policy of protecting the project's intellectual properties are not in place [OWASP, 2016]. Therefore obfuscating JavaScript code of React Native applications is not a universal requirement for secure React Native applications.

4.4. Using components with known vulnerabilities

Nowadays, around 80 percent of JavaScript applications code comes from different libraries and frameworks. Around one-fourth of those libraries added to projects has known to have publicly disclosed security vulnerabilities [Decan, et al 2018]. Nowadays, JavaScript environments package management systems have become very steep on different trivial packages and transitional dependencies. The average JavaScript package's transitional dependency quantity has increased steadily. Nowadays that amount is 80, and a single project can have dozens of unique packages of its own, requiring these transitional dependencies [Zimmermann, *et al* 2019].

So, in JavaScript projects scanning dependencies for known security vulnerabilities should be conducted. Adding a significant number of dependencies without good knowledge about the group responsible for maintaining the project should be avoided.

Having dependency with known vulnerabilities in the application makes the application easy to exploit.

4.4.1 Scanning components with known vulnerabilities

Vulnerabilities on these components can be divided into two categories zero-day vulnerabilities and known vulnerabilities. Zero-day vulnerabilities are undisclosed vulnerabilities, which are exploited less frequently and are much harder to detect [Zimmermann, *et al* 2019]. Detection of these zero-day vulnerabilities would require reviewing packages one by one. This zero-day vulnerability reviewing of packages will be left outside this study's scope. This part of testing is limited to only scanning dependencies with known and disclosed vulnerabilities.

Scanning these disclosed vulnerabilities can be done using package version information to fetch active vulnerabilities from that package with the selected version number. This data is fetched from a community-maintained vulnerability registry. In this study, two tools are used to conduct vulnerability scanning Yarn audit and GitHub security advisors.

5 OWASP mobile Risks

OWASP (Open Web Application Security) is a non-profit foundation not controlled by any single corporate entity. OWASP also publishes all of its research and code freely to the community [OWASP, 2021]. The main reason for OWASPS's existence is to help improve the security in the software industry and educate the community about security risks and prevention methods of those risks in different areas of software development. OWASP does this with the help of tens of thousands of members worldwide. [OWASP, 2020]

This study's core is to research the security testing methodology of mobile applications. This is done more precisely, from React Native client's point of view. So high-level risk categorization mainly used by this study is a list addressing risks in native applications. This is because the main part of React Native applications attack surface is composed of native threats. This study categorizes these native risks by using a list called OWASP Mobile Top Ten. This list categorizes the ten most common risks manifested in the natively running mobile applications life cycle. According to OWASP [2019], the List is composed using publicly gathered information about the most common vulnerabilities found in mobile applications. This was done by using the questionnaire to people who had been subscribed to the OWASP Mobile Projects newsletter.

RISK	Threat Agents	Attack Vectors	Security Weakness		Impacts		Score
	Exploitability	Prevalence	Detectability	Technical	Business		
M1:2016-Improper Platform Usage	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
M2:2016-Insecure Data Storage	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
M3:2016-Insecure Communication	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
M4:2016-Insecure Authentication	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
M5:2016-Insufficient Cryptography	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
M6:2016-Insecure Authorization	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
M7:2016-Client Code Quality	App Specific	DIFFICULT: 1	COMMON: 2	DIFFICULT: 1	MODERATE: 2	App Specific	2.7
M8:2016-Code Tampering	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0
M9:2016-Reverse Engineering	App Specific	EASY: 3	COMMON: 2	EASY: 3	MODERATE: 2	App Specific	5.3
M10:2016-Extraneous Functionality	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0

Figure 7 OWASP mobile risks with assigned severities [Borja, *et al* 2021]

Risk listing created in 2016, is still at the time of writing, the newest mobile risk listing published by OWASP. This listing is still nowadays being used and referenced in

scientific studies and popular literature regarding the security of mobile applications. Examples of such studies are Alanda and others [2020] and Borja and others [2021]. Therefore this risk listing is chosen for the base of our current study and will be used to help determine highly risky areas of React Native applications. More recent material and studies like OWASP MSTG will be used when more precise risks or vulnerabilities are studied.

Risk listing is displayed as a whole in Figure 7 and it includes different areas of severity for each risk. These areas are the difficulty level of exploiting the risk, detectability of the risk, and technical risks severity on the scale of 1-3. In the last column overall score for risk is represented, which is calculated by using scores in previous columns. This score will imply the general risk that risk is causing to a mobile application, and it can be used as a starting point when prioritizing, searching, and fixing found vulnerabilities.

5.1 Improper Platform Usage

When developing either native or hybrid mobile applications, it is necessary to consider that different platforms have their own intended qualities, functionalities, and security features [Facebook A, 2021]. That becomes even more prevalent when hybrid applications are developed with hybrid frameworks like React Native. That is due to the nature of hybrid development. There are now multiple targeted platforms with a single codebase and possibly differently intended functionalities with similar features with usually single team developing the application. That will give the team maintaining and developing the application the responsibility to have knowledge about all targeted platforms and hybrid framework in addition of the native environments themselves [Facebook, 2019]. That would mean in this study's context requirement for the team to having knowledge about four distinct platforms iOS, Android, React-Native, and React.

As stated in Figure 7, technical and business impacts of these vulnerabilities are highly application specific and can vary greatly due to the vastness of Android and iOS platforms, each with thousands of different APIs to exploit [OWASP Top Ten, 2016]. One common issue where the platform can be misused is React-Natives and its platform-specific implementations of AsyncStorage. According to Facebook [2019] AsyncStorage is meant to be used for persisting non-sensitive data in an unencrypted fashion over application restarts [Facebook A, 2021]. However, this unencrypted nature of AsyncStorage can easily be forgotten during development. That can be a problem for example when Redux is used. This problem with Redux can occur because, Redux is used to persist the state of the application to the device, and it can be extremely easy for sensitive data to flow in Redux State [Facebook, 2019]. After sensitive data will flow to persisted, Redux state, data is readily available in readable form even after deletion of data by using iTunes backups. Above stated or many other improper platform usage-

related vulnerabilities can be executed by using malware, which causes the risk to capitalize for a mass of devices that can have critical impacts on all stakeholders [OWASP, 2016].

5.1.1 Testing React Native improper platform usage

As stated in the React Native specific security issues chapter, there is no automatic solution for testing improper platform usage caused by leaky React Native abstraction risk. Therefore, the recommendation of this study is to test leaky abstraction-related improper platform usage by reviewing the use of native APIs by React Native and comparing their respective Android and iOS implementations.

There are static and dynamic analysis tools for analyzing improper platform usage at the native level. These tools like MobFX are further presented in the tools section of the study. However, there are currently no tools found which are made for scanning JavaScript level improper platform usage of React Native.

5.2 Insecure Data Storage

The protection of sensitive data like passwords, medical information, and authentication tokens should be prioritized in all software development projects where sensitive data is stored or processed. Not persisting data at all on the client's level would efficiently mitigate this insecure data storage risk [Mueller, *et al* 2020]. However, this is not usually possible due to practical reasons and would mean in some situations that where user would be forced to enter the complex combination of email and password between every app restart. So, in the mobile environment, most applications with authentication functionality will cache at least some long-duration authentication token that needs to be stored safely away from the possible attackers [Mueller, *et al* 2020].

Insecure data storage risk to capitalize mobile device would have to be accessed physically by a malicious user, accessed with malware or another malicious application spying devices inter applications communication or watching devices clipboard actively. If the phone is physically accessed by a malicious user phone would be connected to the computer. After, connection to the computer, malicious user uses freely available software to access phones file system, including users' sensitive data exposed by insecure data storage risk. Like in other risks phone doesn't have to be physically accessible by the attacker. The attack can be automatized to a larger group of users by using malware and targeting a group of applications known to have issues with exposing sensitive data. This issue is far more prevalent in the mobile environment than in example on web due to the nature of mobile phones being constantly carried around, and lost phone is far more common than in example desktop [OWASP, 2016]. So, during the development phase it should be expected that malicious users are able to inspect unencrypted data storages and will be actively viewing all unencrypted data persisted in device [OWASP, 2016].

As stated in Figure 7 business and technical impacts of these attacks can be varying depending on the data storage solution and type of data stored by the application. This risk would be vastly different in a single-player mobile game where this risk can even be bypassed due to application not storing sensitive data at all [OWASP, 2016]. The opposite would be medical applications, where applications have a realistic possibility of storing social security numbers, medical records, and customers credit card information. In the higher level according to OWASP technical impacts could include identity theft, privacy violation and fraud [OWASP, 2016].

5.2.1 Testing React Native insecure data storage

React Native itself does not make it easy to persist sensitive information to the device securely. That is because there is no module included in React Native implementing safe storage of sensitive information [Facebook, 2019]. Therefore all sensitive data stored in the device should be stored by using a library implementing iOS Keychain, and Android Secure shared storage or encrypted in other means. According to Facebook [2019], One common pitfall is AsyncStorage which many applications are using as persisting Redux state, and sensitive data can flow easily. A final unsafe storing method is React Natives deep linking. Deep linking is used to access the specific application state through the link opened in the mobile device. Information transmitted through deep linking is readable by malicious users and other applications. [Facebook, 2019]

Safe storage method	Unsafe storage method
iOS KeyChain	Deep links
Android Keystore	AsyncStorage
Android Secure Shared Preferences	Android Shared preferences
	Environment variables like React-Native config

Table 3. Checklist for unsafe and safe React Native storage schemes

To test insecure data storage, there are automatic statical tools created like for example MobFX. However, some manual verification with manual identification of sensitive data should also be made [Mueller, *et al* 2020]. This testing can be done in native level by using the steps provided by OWASP [2020]. Unencrypted and encrypted data storage solutions are composed in Table 3.

5.3 Insecure Communication

In mobile applications, there is common practice that SSL or TLS may be used during the authentication but not elsewhere [OWASP, 2016]. This irregular use of secure and insecure communication methods will open different possibilities for attackers. When secure SSL or TLS protocols are not used, attackers can freely intercept and edit the communication between parties anywhere between communicating sides. So more generally, insecure communication risk can be applied when data is transported from point A to point B, and communications get intercepted or eavesdropped. This attack is visualized in Figure 8. Most usual and easiest exploit of this risk would be when data is transported between device and server using HTTP request without proper protection [OWASP, 2016].

Business and technical impacts are also in this case varying, when data is intercepted while it's traversing to the destination it can cause individual sensitive data to be leaked with varying consequences or in worst case scenario admin account being compromised and whole application and systems around it can be compromised. [OWASP, 2016]

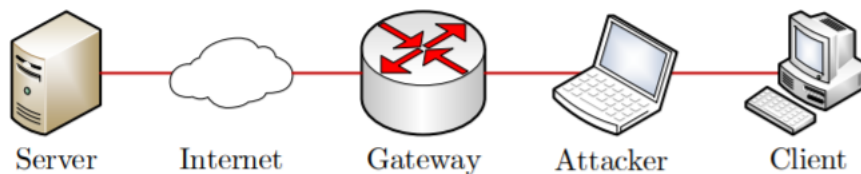


Figure 8 Man-in-the-middle attack visualized [Vondráček, et al 2018].

In previous chapters, communication is mainly defined as HTTP traffic. However, a possibility for insecure communication risk exists with all communication protocols and layers where data is transmitted or received by the device [OWASP, 2016]. In this study's context, only HTTP traffic, including WebSocket connection, will be used and examined. That will make additional protocols like Bluetooth, audio, and NFC out of this study's scope.

5.3.1 Testing React Native insecure data storage

The malicious user would use applications insecure communication vulnerabilities to launch man in the middle attack. In man-in-the-middle-attack, a malicious user is located between two intended participants of the communication. A malicious user would be eavesdropping and possibly modifying the communication between these two intended parties in an unauthorized fashion [Vondráček, et al 2018]. This malicious user can be modeled by using a proxy interception proxy like Burp Suite.

However, an interception proxy at the testing phase is not needed. There are automatic analysis tools included on the tool MobSF that can detect usage of insecure communication methods and SSL pinning vulnerabilities. Therefore primary analysis of

this risk is done by MobSF. Interception proxy like Burp Suite is used only if a vulnerability is found, and penetration testing needs to be conducted.

5.4 Insecure Authentication

In order to Insecure authentication vulnerability to be exploitable at the application-level, malicious user should be able to execute functionality or access data that is not intended to be executed or accessed user without proper login credentials. This execution can happen within the mobile client or directly in the backend service [OWASP, 2016].

According to OWASP Top Ten listing [2016], there are two common scenarios for a malicious user to bypass authentication. These bypassing methods are attacks against offline and online authentication process of applications. In offline authentication bypassing applications, local binaries are exploited by using binary attacks. Binary attacks aim to force applications to skip the local authentication process and execute an action as an anonymous user without proper login credentials [OWASP, 2016]. The second part of the risk is bypassing the online authentication process. In online authentication bypassing malicious user exploits authentication scheme to execute unprivileged actions in backend systems. These can happen by sending forged POST or GET HTTP requests to the backend without proper credentials.

Insecure authentications impacts can be technically critical as the system cannot identify the user performing the actions. That can add additional problems in top of unprivileged actions executed. Not identifying the user executing these overprivileged actions can delay detection and identifying exploited security vulnerabilities. That will make it harder to prevent future attacks against the system [OWASP, 2016].

5.4.1 Testing for insecure authentication

General OWASP-based security scanning tools like MobSF will expose some vulnerabilities in the applications authentication scheme. However, there are a lot of different authentication methods, including custom ones; therefore, there is no one-catch-all solution for testing the security of all authentication schemes [Mueller, *et al* 2020]. OWASP MSTG defines testing methodologies for different authentication schemes like OAuth and 2-Factor Authentication. Therefore, an additional MASVS checklist, visible in Figure 9 will be used during this study as additional help for testing the authentication process.

V4	Authentication and Session Management			
4.1	MSTG-AUTH-1	If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.	✓	✓
4.2	MSTG-AUTH-2	If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user's credentials.	✓	✓
4.3	MSTG-AUTH-3	If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm.	✓	✓
4.4	MSTG-AUTH-4	The remote endpoint terminates the existing session when the user logs out.		
4.5	MSTG-AUTH-5	A password policy exists and is enforced at the remote endpoint.	✓	✓
4.6	MSTG-AUTH-6	The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.	✓	✓
4.7	MSTG-AUTH-7	Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire.	✓	✓
4.8	MSTG-AUTH-8	Biometric authentication, if any, is not event-bound (i.e. using an API that simply returns "true" or "false"). Instead, it is based on unlocking the keychain/keystore.		✓
4.9	MSTG-AUTH-9	A second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced.		✓
4.10	MSTG-AUTH-10	Sensitive transactions require step-up authentication.		✓
4.11	MSTG-AUTH-11	The app informs the user of all sensitive activities with their account. Users are able to view a list of devices, view contextual information (IP address, location, etc.), and to block specific devices.		✓
4.12	MSTG-AUTH-12	Authorization models should be defined and enforced at the remote endpoint.	✓	✓

Figure 9 OWASP MASVS checklist for authentication

5.5 Insufficient Cryptography

At the highest level, in order to insufficient cryptography risk to capitalize, a malicious user would have to be able to have access encrypted data as plaintext in an unauthorized fashion. This risk can be categorized into two main categories. First category of risk is the situation where encryption and decryption functionality is fundamentally flawed, and functionality can be exploited to gain access to the plaintext of sensitive data [OWASP, 2016]. In the first category, up to standard encryption algorithms are used, but the application can store the private key of encryption in the readable form inside applications storage. In this scenario, an attacker uses a binary attack against the application to gain access to a private key which causes mobile operating-system-level cryptography to become compromised. A completely flawed cryptographical system causes attackers to gain access to application data without the need for brute-forcing and a large number of computations.

The second category of this risk is the use of insecure and deprecated algorithms which are not up to date for modern computational and security standards [OWASP, 2016]. Example about unmodern algorithm would-be situation where MD5 or SHA1 hashing functions are used. These one-way hashing functions have been previously used as a way of storing user passwords. Previously it was thought to be computationally infeasible to revert hashing functions like MD5, back to the original string. However, nowadays, these hashing functions are perceived as unsafe methods for protecting sensitive data [Bhandari, *et al*, 2017]. That is due to the way of converting globally same strings to identical hashes. That creates a possible attack vector that is either brute-forcing or using lookup tables openly published in the internet where millions of hashes and plaintext pairs are stored and easily reversible. Outdated or insecure cryptography methods are not only limited to obsolete hashing functions. The second example is Apple's iOS application protection. Apple requires all iOS published in App Store to be signed by a trusted source in order to the application to be executable in non-jailbroken iOS devices. That should, in theory, prevent users from conducting the binary attacks

against the application however, according to OWASP [2016], freely available tools are able to circumvent this encryption by taking snapshots of the decrypted app after iOS launches application to devices memory [OWASP, 2016].

Technical and business impacts of insufficient cryptography risks vary depending on data that is protected by cryptography. Also, a way that the cryptosystem is flawed causes very different risks. Where either single entry can be exposed when a single device is compromised or in a situation of leakage of shared private key all customer data encrypted using that private key. In the end, in business sense, retrieval of plain text form of encrypted data will cause intellectual property theft, privacy violations, and exposure of sensitive data [OWASP, 2016].

5.5.1 Testing for insecure cryptography

In the scope of this study first part of the cryptographic risk completely flawed cryptographic system is researched in the stage where insecure storage is studied. That would mean searching for private keys from application packages. The second part of the insecure or deprecated algorithm is studied by using automatic statical and dynamic analysis tools, mainly MobSF, which can detect usage of deprecated cryptographic functions like MD5 or SHA.

5.6 Insecure Authorization

This risk is similar to insecure authentication risk, and the difference between these can be confusing. In insecure authentication risk, malicious users were accessing APIs or offline authentication using non-valid login credentials or as fully anonymous users. So this causes the attacker to pass the whole authentication process. In this insecure authorization risk, the malicious user passes the authentication process as intended. After successful authentication, the user will forcefully browse to a vulnerable endpoint and execute higher privileged functionality as authenticated user is meant to. These kinds of attacks are usually made by using stolen accounts and mobile malware. The main difference to insecure authentication is that auditioning of user and executed functionality is generated. [OWASP, 2016]

Technical and business impacts of insecure authorization are similar to insecure authentication when exploited. The difference is that in this risk, logging is usually conducted properly, and therefore insecure authorization vulnerabilities are easier to detect when exploited [OWASP, 2016].

5.6.1 Scanning insecure authorization

Methodology for testing insecure authorization and authentication vulnerabilities are defined under the same generic authentication architecture section in Mobile Application Security Testing Guide [2021] as Insecure authentication risk, and testing will happen

similarly as testing insecure authorization in 5.4. Therefore, testing authentication architecture as a whole is explained more detail in the chapter where testing insecure authentication is presented.

5.7 Poor Code Quality

Code quality risk refers to all bugs or vulnerabilities exposed by bad coding conventions or mistakes in mobile projects that are done at the code level and therefore are easily fixable. This risk includes typical vulnerabilities like buffer overflows, XSS, CSRF, and SQL injection, which will cause foreign code execution [OWASP, 2016].

Code quality issues are common within mobile code. However, all code quality issues are not actual problems because in many cases, code quality issues commonly cause benign vulnerabilities that will not cause actual damage. In addition of that, code quality issues are commonly hard to detect only by reviewing applications binaries. Therefore, security testers are using automated statical source code analysis tools to detect code quality issues by using either application's source code or package. [OWASP, 2016]

Like in other OWASP Mobile risks impacts of these code quality issues are application, business, and exploit specific. This means that impacts can vary with different severity levels technically and in businesswise ranging from performance degradation by denial-of-service attack to sensitive data exposure by using SQL injection [OWASP, 2016].

5.7.1 Finding Code Quality issues

Code quality issues should be studied and are prevalent in different code levels of React Native applications. In addition to pure native risk, React Native-specific risks are suggested in chapter 4, where bad JavaScript code quality and scanning components with known vulnerabilities are reviewed more thoroughly. These all issues should be tested in all levels of React Native applications code, including different Native environments.

Finding code quality issues is done mainly automatically by using different static scanners [OWASP, 2016]. One exception to that automation is build settings that are beneficial to review at least semi-manually, at least in situations when applications haven't gone through previous thorough security testing [Boduch, et al. 2020]. This can be done by using the platform-specific code quality checklist in OWASP MSTG.

5.7.1.1 JavaScript

The first platform which is reviewed is JavaScript and React Native specific code quality scanning. There are two possible surface areas of attack using code quality-related vulnerabilities in the JavaScript environment. React Native related hybrid configurations and JavaScript code logic. There is a large market of widely used JavaScript static

analysis tools. Most of these are used for helping the team to keep coding conventions consistent. Examples of tools with said functionality are ESLint and Prettier. One such tool that has vulnerability scanning functionality included is SonarQube. According to its documentation, it is an automatic code review tool to detect bugs, vulnerabilities, and security hotspots in the JavaScript codebase [SonarQube, 2021]. SonarQube is used as the tool for scanning the JavaScript parts of an application.

5.7.1.2 Native environments

Like in JavaScript environments, there are two possible surface areas to attack native iOS and Android ecosystems. These are platform-specific build configurations and platform-specific code itself. Starting from build configurations, automatic ways of validating the security of these configurations are IDEs iOS IDE XCode and Android IDE Android studio. [Boduch, *et al*, 2020].

React Native would not have any platform-specific code in a perfect world, and all custom code is written using platform-agnostic JavaScript; therefore, this section would not be needed. However, this is not usually the case, and therefore tools for validating different native environments by code quality are selected. SonarQube is used for testing JavaScript for code quality issues, but it also has support for native platform-specific languages of React Native, therefore it is used for also scanning native environments. In addition, our generic security reviewing tool MobSF will check code quality issues from native environments. Therefore, no additional code quality-specific tools are needed for this section.

5.8 Code Tampering

After reverse-engineering the application's source code, one possible direction of attack is code tampering. In code tampering attack malicious user is modifying the applications binaries or resources and running the modified package on device. Code tampering can be divided into two main categories. First of these categories is binary patching where the existing binary of application is modified. That happens when a malicious user opens the application in a binary hex editor and decompiles the application. After the application has been successfully decompiled, the application's binary is edited and that application is repackaged and redistributed. This way of modification had been more common and easier in the past and has become much harder nowadays due to application signing requirements in the Android and iOS application stores. This Code signing functionality can usually be circumvented easily. This circumvention requires the user user to disable the verification of the applications package before the tampered application can be executed in target device [Boduch, *et al*, 2020].

Second common method of code tampering is the tampering of applications process memory during the execution of an application. These methods and tools to tamper

application process memory are widely and freely available, and the code injection process is much harder to detect and prevent from the application's point of view [Boduch, et al, 2020].

Business and technical impacts like in many other risks can vary depending on the target application. However, according to OWASP [2016], there are two common types of impacts on business when conducting binary tampering. The goal of the first attack would be to steal the application's source code and repackage the application for redistribution. Mobile games are particularly popular targets for this type of code tampering where, paywalls of freemium games are removed. After removing the paywall, game is redistributed to users with possibly inserted malware, which will most likely impact the game creator's revenue and reputation.

Second type of tampering would target applications containing sensitive information, in example application like a banking application would be targeted and a counterfeit version of the application is created and redistributed like in the previous mobile game example. This type of attack has a difference to the previous attack, which is that new application is marketed to the user as a legitimate banking application and when a user is logging in to the malicious clone application, login credentials will be stolen and used to create fraud or identity theft. [OWASP, 2016]

5.8.1 Testing for Code tampering

Code tampering and reverse engineering requirements for applications are similar. Both are based on the same set of methodologies under section Anti-RE of Android and iOS applications in OWASP MASVS [2019]. Therefore, testing reverse engineering and code tampering is defined more in detail in the next chapter, where these reverse engineering testing methodologies are introduced.

5.9 Reverse Engineering

First step of attacking for attacker or security tester would be to gather more information about the application exploited [Palacios, *et al*, 2019]. One of the best ways to gather information about mobile application would be to find how the application functionally works. That can be done best when source code of an application can be researched. That is one of the reasons why many applications will have some level of binary protection implemented to mobile applications. However, this binary protection can be relatively easily circumvented by loading the target application to a freely available de-obfuscation tool, which is why obfuscation should never be the only security control for mobile applications [Mueller, *et al* 2020]. When the application is loaded to the de-obfuscation tool and the program's source code is revealed successfully to the malicious user, the next step would be to analyze the application's source code for vulnerabilities by using different statical and dynamical analysis tools. The absolute worst-case scenario with

reverse engineering would mean leakage of the database connection string, the private key of encryption which would be commented in this scenario directly to applications source code.

Application reverse engineer and binary protections have similar properties where cryptography and reverse engineering is a constant race between attackers and developers. According to OWASP [2016] mobile environment is particularly vulnerable to reverse engineering attacks due to the nature of its code being dynamically inspectable. So similarly, as in cryptography role of obfuscation of the code is to make the flow of the application hard as possible to understand and inconvenient as possible to de-obfuscate.

Business threats of reverse engineering can also vary greatly depending on the type of application de-obfuscated. According to OWASP [2016], the most usual business risks would be identity theft, intellectual property theft, and reputational damage. However, technical impacts might be more unpredictable as an attacker could steal the application's source code and deploy fake application versions. That would then allow the attacker to gain access and steal unsuspecting user login credentials. In most cases, the attacker would use reverse engineering to gain intelligence for starting follow-up attacks, including code tampering, revealing cryptographic constants, and performing attacks against backend systems [OWASP, 2016].

5.9.1 Testing for reverse engineering

Testing reverse engineering in this study consists of two parts. First part is to test JavaScript obfuscation defined more precisely when React Native specific vulnerabilities listing in chapter 4. This will happen by inspecting projects main bundle in iOS and Android devices and checking whatever JavaScript bundle has been obfuscated correctly.

Second part of testing process is to test reverse engineering requirements like testing would happen in any native application. MobSF has the module for testing reverse engineering requirements [Joseph, *et al*, 2021]. Therefore, initial testing of reverse engineering is left to MobSF. If the tool finds reverse engineering vulnerabilities from application penetration testing tools like APKTool will be used in the penetration testing phase.

5.10 Extraneous functionality

In extraneous functionality, a malicious user will download the application to their own local environment and will examine log files, binaries, and configurations to find hidden functionalities that are not meant to be executed or are left to the application by accident. This risky hidden functionality commonly can be development bypasses or development functionalities left to production application by accident [OWASP, 2016]

According to OWASP [2016], it is common practice in mobile development that applications will have different authentication methods, admin control panels, or logging

systems used in production than in development and staging environments. Therefore, extraneous functionality risk is so prevalent with mobile applications. More interestingly, according to OWASP [2016], there is a high likelihood that any given application will have some extraneous functionality not intentionally exposed to the user but is still accessible with the right tools. However, this does not necessarily mean that most of the applications are vulnerable to extraneous functionality risk. In many cases, these extraneous functionalities are not harmful and can be benign in nature and in this sense, unusable to a malicious user.

In the end, business threats of these harmful extraneous functionalities are business and application-specific. These can differ from the worst-case scenario where malicious users gain access to freely execute high-privileged actions or bypass authentication in a banking application. The opposite situation would be with the offline Android mobile application, which exposes logs in the production environment by adding `debug=true` flag manually to the locally installed version of the application. [OWASP, 2016]

5.10.1 Testing for extraneous functionality

Detecting extraneous functionality automatically usually happens by using string table analysis where applications code is searched for known harmful extraneous functionalities commonly left in application [OWASP, 2016]. MobSF has functionality for conducting string table analysis of extraneous functionality [MobSF, 2016]. The framework includes the functionality of checking if an application allows adding `debug=true` flag to production build and exposing extraneous functionality in that way.

6 Mobile security testing tools

Mobile environments penetration testing and vulnerability assessment tools can be divided into two categories. First category is static analysis tools. These tools are used to automatically scan the source code and package of the application without running the application. Therefore static analysis of applications will find vulnerabilities primarily by searching for the usage of insecure APIs, problems with applications configurations and code quality issues like buffer overflow [Bojjagani, *et al*, 2017]. Static analysis is considered a highly scalable solution with the possibility of including a static analysis tool to the IDE or CI pipeline, which is making it a powerful tool for identifying possible vulnerabilities already at the development phase [Dewhurst 2020]. As a tradeoff for being highly scalable static analysis tools can generate a large number of false positives due to tools' low understanding of applications' context. An example of this could be web-related CSRF and XSS vulnerabilities, which are rarely issues in mobile applications. However, web-based analysis tools can be used to detect them from mobile applications [Boduch, *et al*. 2020].

The second category of security testing tools is dynamic analysis tools. These tools are used when applications are at a runtime state. Where static analysis was scanning vulnerabilities from the source code, dynamic analysis is used to find issues from server configurations, authentication, authorization, and overall data flow of the application by making real requests to backend systems [Boduch, *et al*. 2020].

6.1 MobSF

MobSF is an open-source and free to use penetration testing, malware analysis, and security assessment framework used for detecting OWASP Mobile Top risks from mobile applications. According to MobSF [2021] framework is an all-in-one mobile application testing solution targeting all popular mobile platforms, including Android, iOS, and Windows. The tool is used by multiple recent studies as the primary tool for detecting vulnerabilities [Kohli, Narmada and Mahsa, 2020] and [Hatamian, *et al*, 2021]. The result of the static analysis, by using the MobSf and intentionally vulnerable application can be seen in Figure 10.

MobSF has tools for analyzing mobile applications both statically and dynamically. Statical analysis tool of MobSF uses lists of keywords known as string table. These vulnerabilities included in this string table can be for example deprecated cryptographic algorithms and usage of outdated native APIs. This method can cause the tool to report a large amount of false positives when conducting statical analysis [Hatamian, *et al*, 2021]. Therefore, a manual review of tools statically detected vulnerabilities should be conducted.

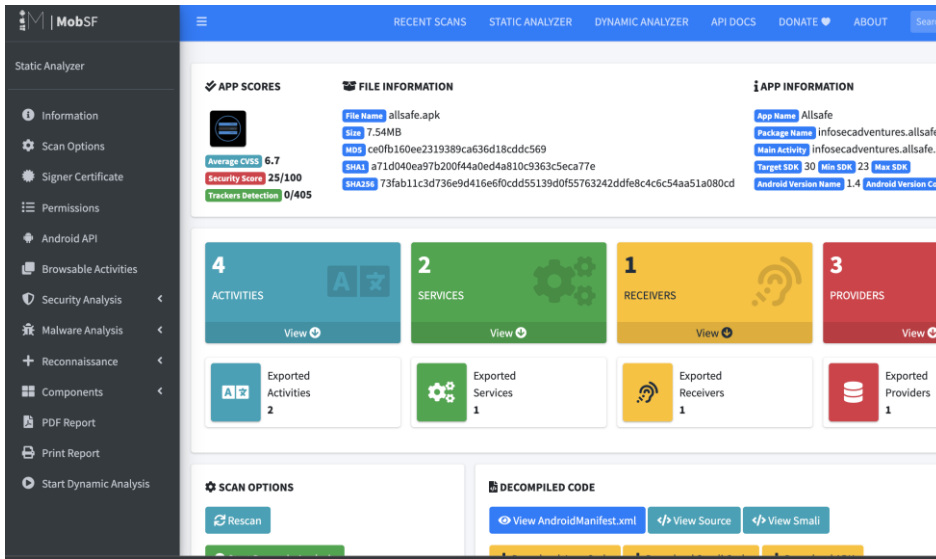


Figure 10 Result of MobSF statical scan using intentionally vulnerable allsafe.apk <https://github.com/i0thkr1s/allsafe>

6.2 Burp Suite

Burp Suite is a security testing toolset created by PortSwigger. Burp Suite includes a wide variety of different tools including interception proxy, intruder, and repeater [Burp, 2021]. In this study, Burp Suites interception proxy is used to conduct passive and active MITM-Attacks. Where active attacks has the goal of editing data of communication between the parties, passives is only used to observe the data. This functionality of the tool in user interface level is illustrated in Figure 11, where the interception functionality of Burp Suite is displayed [Burp, 2021].

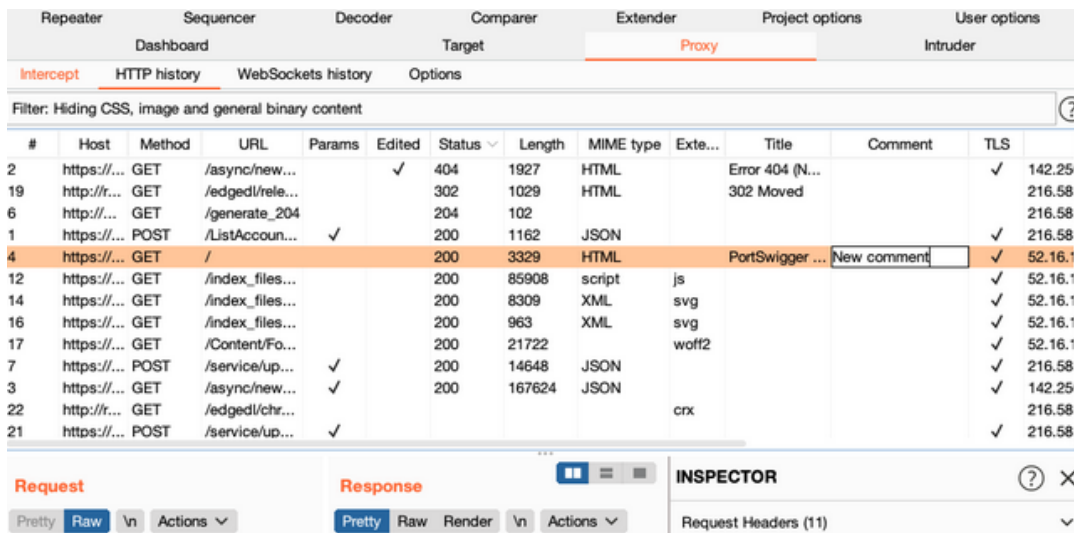


Figure 11 Burp suite tool actively listening HTTP-traffic [Burp, 2021].

Most interception proxies, including Burp Suite, supports only interception of HTTP and HTTPS communication protocols, which includes WebSocket communications [Boduch, *et al.* 2020]. However, that excludes some communications used commonly by

mobile applications like notifications, Bluetooth, and NFC which can be visualized and intercepted using plugins developed to Burp Suite [Burp, 2021].

6.3 SonarQube

Security testing tool MobSF is used for analyzing Android APKs and iOS IPAs and their native parts. However, this is not enough when testing an application created using React Native. In React Native application, logic of the application is executed inside JavaScript virtual machine. Therefore, tool like SonarQube is needed for statically analyzing applications' JavaScript source code. The result of SonarQube's statical scan targeting unmaintained React Native application can be seen in Figure 12.

SonarQube is one of the most used statical code and vulnerability analysis tool adopted by both academia and the software industry [Lenarduzzi, *et al*, 2019]. In the time of writing, the tool supports 26 different programming languages [SonarQube, 2021]. Most importantly, for React Native applications the tool supports scanning of platform-agnostic JavaScript and platform-specific languages like Java, Kotlin, Swift and Objective-C.

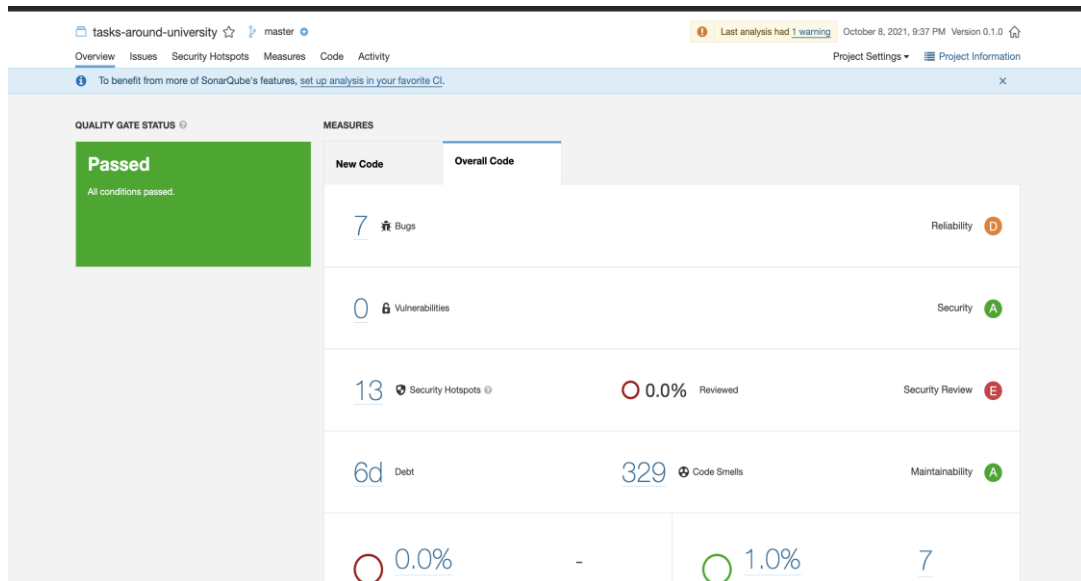


Figure 12 Result of SonarQube scan using outdated React Native application <https://github.com/jalijuhola/tasks-around-tampere>

6.4 ApkTool

There are multiple tools doing seemingly the same functionality of reversing Android APKs to readable Java source code. However, only one such tool is needed for the study. So Arnativichses and others' [2018] study where different Android reverse engineering tools were compared was selected. Main metric of the study's comparison was reverse engineering transformation accuracy of different tools. At the end, tool called ApkTool

was by far the most accurate tool for reversing Android APKs. In addition of that according to Arnativichses and others [2018], ApkTool has also the functionality of conducting code tampering attacks and reassembling the tampered packages. Therefore, ApkTool was selected as a tool for conducting reverse engineering and code tampering attacks during the case study.

6.5 GitHub Security Advisories

GitHub security advisors is a tool used for scanning the security of dependencies on a project published into repository hosting service GitHub. GitHub security advisors is not mobile or JavaScript-specific tool, but it is designed to be a universal tool for supporting all technologies, which have vulnerabilities added to cve.mitre.org vulnerability identification system [GitHub, 2021]. Therefore, tool can be used for finding vulnerabilities from React Native projects in all different environments including JavaScript, Java, Objective-C, and Swift. Results of GitHub Security advisors targeting outdated React Native project and its Python-based backend implementation can be seen in Figure 13.

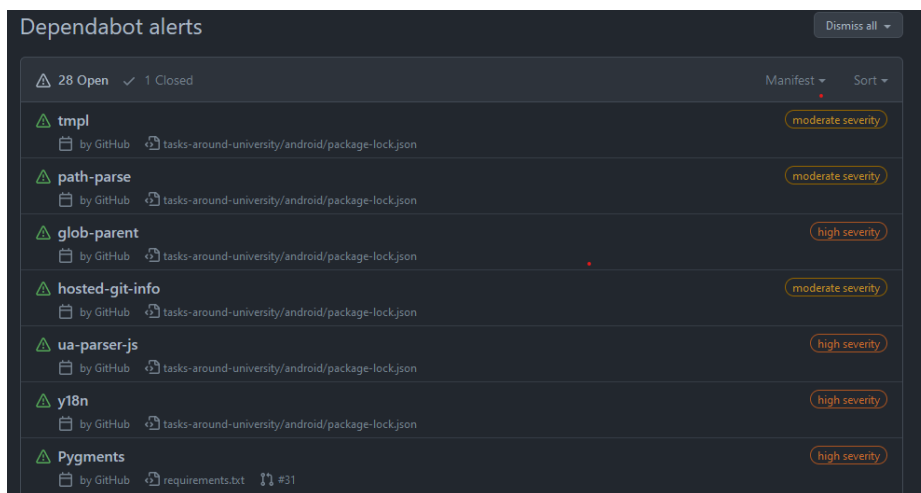
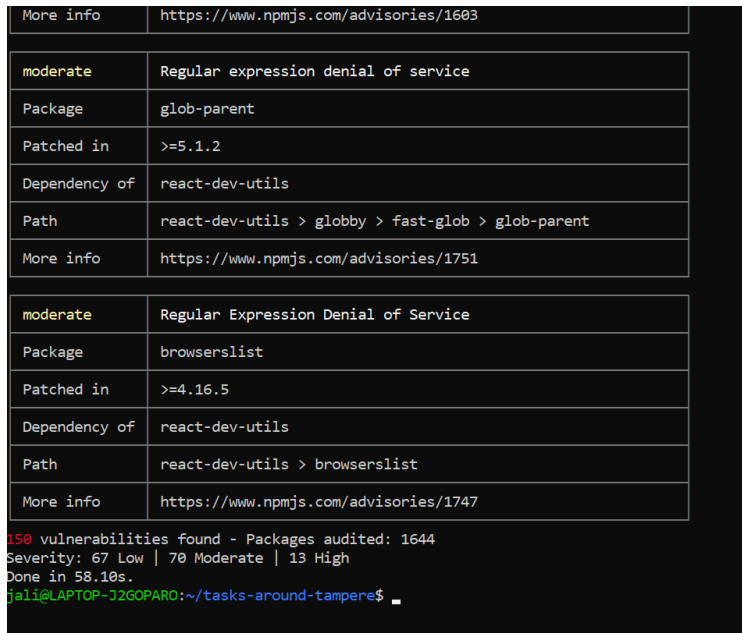


Figure 13 GitHub Security advisories of old and outdated react native project <https://github.com/JaliJuhola/tasks-around-tampere>.

For package maintainers and admins, GitHub security advisors will show vulnerabilities found by scanning the repository’s dependencies. That makes it possible to keep vulnerabilities confidential until a patch is made. After the patch to fix the vulnerability is made vulnerability can be revealed to the public, and users of the package or project can be recommended to update the project or package [GitHub, 2021]. Project maintainers and admins can also discuss and disclose vulnerabilities about the package they are developing through GitHub security advisors. That, however, is meant more to be used by maintainers of a project and therefore is out of the scope of this study’s context.

6.6 Yarn Audit

The GitHub security advisor's scanner can be only used to scanning vulnerabilities in GitHub repositories. Other standalone tool Yarn Audit is also presented in this study. Yarn audit will be executed by running the command *yarn audit* in the JavaScript projects root. It is used similarly to check for known security issues of installed JavaScript dependencies, an example of the tools output can be seen in Figure 14.



```
More info | https://www.npmjs.com/advisories/1603
-----
|
| moderate | Regular expression denial of service
| Package  | glob-parent
| Patched in | >=5.1.2
| Dependency of | react-dev-utils
| Path      | react-dev-utils > globby > fast-glob > glob-parent
| More info | https://www.npmjs.com/advisories/1751
-----
|
| moderate | Regular Expression Denial of Service
| Package  | browserslist
| Patched in | >=4.16.5
| Dependency of | react-dev-utils
| Path      | react-dev-utils > browserslist
| More info | https://www.npmjs.com/advisories/1747
-----
150 vulnerabilities found - Packages audited: 1644
Severity: 67 Low | 70 Moderate | 13 High
Done in 58.10s.
jali@LAPTOP-J2GOPARO:~/tasks-around-tampere$
```

Figure 14 Yarn audit used to old outdated react native project <https://github.com/JaliJuhola/tasks-around-tampere>.

6.7 Frida

Frida is a command line-based multi-platform dynamic analysis toolkit created by Ole André Vadla Ravnås. Frida is used to dynamically analyze Android and iOS Mobile applications and JavaScript Web applications. In addition to these environments, Frida supports the analysis of other platforms, outside of this study's scope [Mueller, 2020]. Frida has been an open-source project with free to use policy, which allows other tools like Frida toolkit to include Frida's source code as a whole to their project. Frida does this because it as an organization has a goal to enable creation of the next generation security testing tools [Frida, 2021]. This free-to-use policy can be seen in example with MobSF, which is basing on its dynamic analysis of Android applications purely on functionality provided by Frida [MobSF, 2021].

Frida has very wide variety of functionality in it. This includes hundreds of methods included in Frida API [Frida, 2021]. Main parts of tools functionality is tracing applications networking and function calls with the option to intercept these. In addition to this tracing, Frida has a functionality to read and manipulate applications local

filesystem and local databases. Final interesting functionality of Frida is a possibility to read applications' runtime memory and tamper it at runtime. [Frida, 2021]

There is very little automation included in these functionalities provided by Frida. This is why implementing these fully to the testing process would require lot of time and in-depth knowledge about Frida and the different platforms tested. This generic and flexible approach to the testing process would be time-consuming. However, this would make React Native testing process more comprehensive by targeting more low-level React Native APIs. In the future study this would be Having a possibility of building automated React Native testing tools based on Frida. However, In this study multiple tools are used and digging more a single tool like Frida is determined to be out of this study's scope. Therefore, only ready-made Frida snippets published to Frida's CodeShare service are used.

6.8 QARK

Quick Android Review Kit (QARK) is a command-line-based open-source Android security testing tool created by LinkedIn [LinkedIn, 2018]. QARK has functionality for automatically testing the most common platform-specific vulnerabilities of Android. After vulnerability scanning, QARK determines if found vulnerabilities can be generated and generates APK capable of exploiting the found vulnerabilities.

6.9 Insiders

Insiders is a similar multi-platform security testing tool like MobSF. It has functionality of scanning OWASP Top Ten vulnerabilities from the Java, Kotlin, Swift and JavaScript environments [InsiderSec, 2020]. This happens by analyzing the source code of the application similarly as SonarQube analyses Java and Swift code. Insiders is included in the study as SonarQube is not capable of analyzing Swift source code for vulnerabilities and there was a need for a source code analysis tool for Swift.

6.10 OWASP Dependency Checker

OWASP Dependency Checker is a tool executed from the command line interface and it is used for scanning dependencies of applications implemented by different technologies for known vulnerabilities. OWASP Dependency Checker is according to its documentation capable of scanning 14 different types of projects including React Natives native mobile technologies like Maven, Gradle, CocoaPods, and Swift for known vulnerabilities. [Long, 2016]

In this study, JavaScript environment is scanned by using Github Security Advisors and Yarn audit. These tools are lacking the capability for testing native environments like Androids Gradle and iOS CocoaPods. Therefore, in this study, OWASP dependency checker is used for scanning native environments of React Native for known

vulnerabilities. The example output of the tool can be seen in Figure 15 where an outdated project has been scanned for known vulnerabilities.

Summary

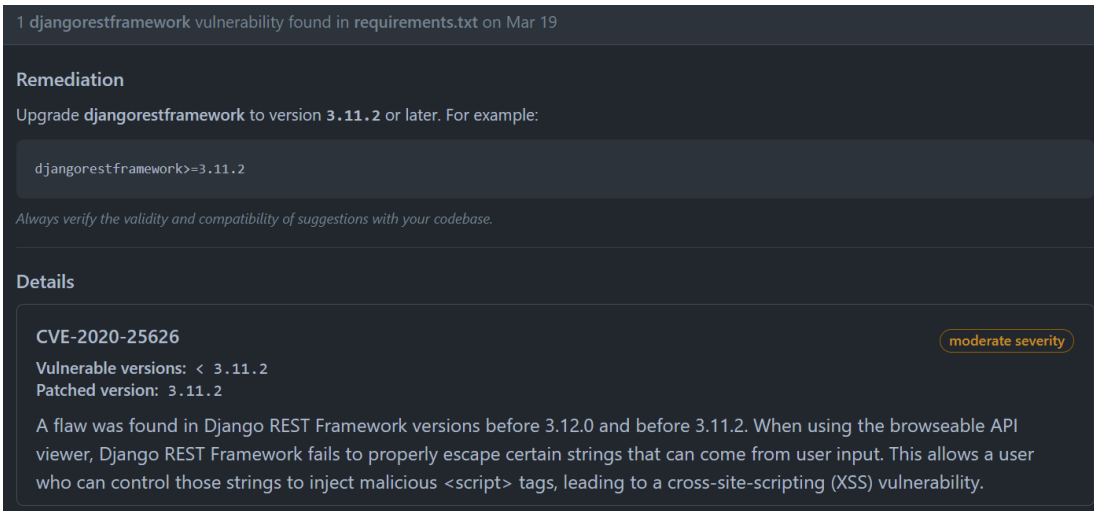
Display: [Showing Vulnerable Dependencies \(click to show all\)](#)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
acorn:6.0.2		pkg:npm/acorn@6.0.2	high	3		3
ansi-regex:3.0.0		pkg:npm/ansi-regex@3.0.0	moderate	1		3
axios:0.18.0		pkg:npm/axios@0.18.0	high	6		3
braces:2.3.2		pkg:npm/braces@2.3.2	low	1		3
color-string:1.5.3		pkg:npm/color-string@1.5.3	moderate	1		3
glob-parent:2.0.0		pkg:npm/glob-parent@2.0.0	HIGH	2		3
handlebars:4.0.12		pkg:npm/handlebars@4.0.12	high	15		3
hosted-git-info:2.7.1		pkg:npm/hosted-git-info@2.7.1	moderate	3		3
ini:1.3.5		pkg:npm/ini@1.3.5	high	2		3
js-yaml:3.12.0		pkg:npm/js-yaml@3.12.0	high	4		3
json-schema:0.2.3		pkg:npm/json-schema@0.2.3	moderate	1		3
kind-of:3.2.2		pkg:npm/kind-of@3.2.2	high	1		3
lodash:3.10.1		pkg:npm/lodash@3.10.1	high	16		3

Figure 15 Example of results of scanning JavaScript environment of outdated React Native Project <https://github.com/JaliJuhola/tasks-around-tampere>.

7 Vulnerability categorization

In this thesis, all vulnerabilities scanned are already categorized under risk groups according to OWASP Top Ten threats and React Native risk listing. So where is the need for the second way of categorizing vulnerabilities differently? If vulnerabilities were only categorized in the risk level, categorization would be really generic and it would be hard to determine the risk and exploitability of a found vulnerability. Exploitability and overall risk of certain disclosed vulnerabilities can be determined by using active discussion and existing research of found vulnerabilities. This is where concepts introduced in this chapter, universal vulnerability databases and identifiers of vulnerabilities are useful. This vulnerability-related data is illustrated in Figure 16 [NIST, 2021].



The screenshot shows a GitHub security advisory for CVE-2020-25626. At the top, it states '1 djangoestframework vulnerability found in requirements.txt on Mar 19'. Under the 'Remediation' section, it instructs to 'Upgrade djangoestframework to version 3.11.2 or later. For example:' followed by a code block containing 'djangoestframework>=3.11.2'. A note below says 'Always verify the validity and compatibility of suggestions with your codebase.' The 'Details' section shows the CVE identifier 'CVE-2020-25626', 'Vulnerable versions: < 3.11.2', and 'Patched version: 3.11.2'. A 'moderate severity' badge is visible. The description reads: 'A flaw was found in Django REST Framework versions before 3.12.0 and before 3.11.2. When using the browsable API viewer, Django REST Framework fails to properly escape certain strings that can come from user input. This allows a user who can control those strings to inject malicious <script> tags, leading to a cross-site-scripting (XSS) vulnerability.'

Figure 16 Vulnerability Mitigation, CVSS score, and CVE identifier provided by GitHub security advisors.

7.1 Common Vulnerability Scoring System

Common Vulnerability Scoring System (CVSS) is a framework created and maintained by First.org. CVSS has the goal of informing characters and severity of found vulnerabilities in different types of software [NIST, 2021]. CVSS scoring system is based on calculating severity scores between zero and ten to vulnerabilities. This is done by using the calculator provided in Figure 17. After the vulnerability score has been calculated, the criticality of vulnerability can be determined using the list provided in Figure 18. These calculated Scores are often used to prioritize the prevention and patching of found vulnerabilities [NIST, 2021].

CVSS v3.0 Ratings

Severity	Base Score Range
None	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

Figure 17 Vulnerability criticality by score [NIST CVSS, 2021].

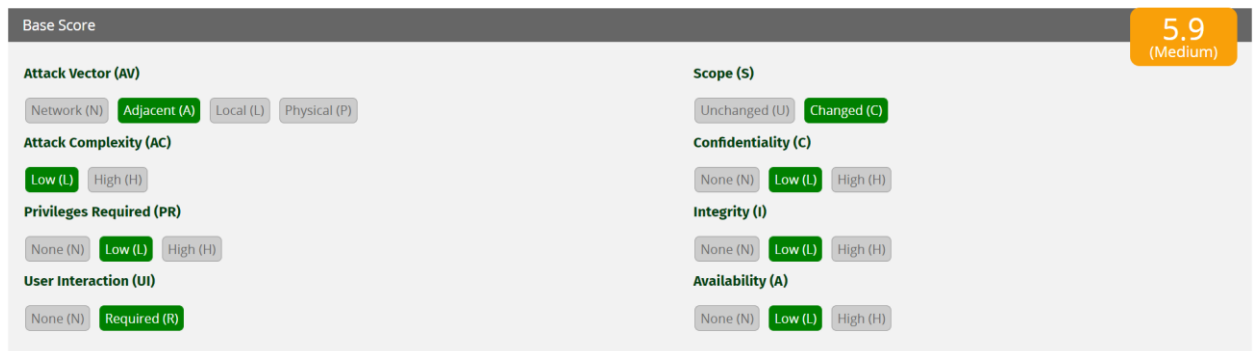


Figure 18 CVSS base scoring system [First.org, 2021]

7.2 National vulnerability database

National Vulnerability Database (NVD) is an open and free-to-use vulnerability management system created by NIST [NIST, 2021]. NVD works with CVE and does not report or study vulnerabilities by itself, but it fetches publicly-listed CVE identifiers from MITRE Corporation. After a new vulnerability has been disclosed by MITRE and delivered to NVD. NVD displays data related to vulnerability like analysis of the vulnerability, CVSS score and vulnerability description on its website. At the beginning, CVE will be added to the public NVD database as an unpatched vulnerability and possible fixes of vulnerability are published after, a fix is found. [NIST, 2021]

🔗 CVE-2020-1896 Detail

Current Description

A stack overflow vulnerability in Facebook Hermes 'builtin apply' prior to commit 86543ac47e59c522976b5632b8b9a2a4583c7d2 (<https://github.com/facebook/hermes/commit/86543ac47e59c522976b5632b8b9a2a4583c7d2>) allows attackers to potentially execute arbitrary code via crafted JavaScript. Note that this is only exploitable if the application using Hermes permits evaluation of untrusted JavaScript. Hence, most React Native applications are not affected.

[+View Analysis Description](#)

The screenshot displays the severity and metrics for CVE-2020-1896. It features a 'Severity' header with tabs for 'CVSS Version 3.x' (selected) and 'CVSS Version 2.0'. Below this, the 'CVSS 3.x Severity and Metrics' section includes a NIST NVD icon, a 'Base Score' of 9.0 CRITICAL, and a 'Vector' of CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H. A note at the bottom states: 'NVD Analysts use publicly available information to associate vector strings and CVSS scores. We also display any CVSS information provided within the CVE List from the CNA. Note: NVD Analysts have published a CVSS score for this CVE based on publicly available information at the time of analysis. The CNA has not provided a score within the CVE List.'

Figure 19 Description and CVSS information of React native vulnerability CVE-2020-1896 [NIST, 2021].

7.3 Common Vulnerabilities and exposures

Common Vulnerabilities and exposures (CVE) is a publicly available vulnerability identification database consisting identification information of publicly disclosed vulnerabilities. It is created and maintained by MITRE Corporation, and it has aim for making addressing vulnerabilities consistent across the software industry. [Mitre, 2021]

According to Mitre [2021], CVE does not provide information about risks, impacts, or ways to fixing the vulnerability by itself. Idea of CVE is to provide only standard vulnerability identifier, and a brief introduction. This information is then used by different vulnerability databases like National Vulnerability Database (NVD) to host information about vulnerability's risks and possible fixes. That allows consistent identification of vulnerabilities between different vulnerability databases.

8 React Native security testing process

Previously in related work chapters Figure 3, different security testing models were compared by using a systematic literature review conducted by Haq and Khan [2021]. This systematic literature review of security testing models is used as a reference for selecting a base for the security testing process created during this study.

The requirements of the security testing model for React Native applications are that model should be compatible with OWASP mobile risks and MSTG methodologies introduced previously. This gives the selected model a requirement that it cannot be too strict or explicitly built for any specific native or hybrid technology. This will allow customizing the model according to React Native specific testing needs defined in previous chapters. The second requirement for the model is that it supports testing native environment-specific risks in both Android and iOS environments.

Models that were chosen for comparison from Haq's and Khan's [2021] systematic literature review are OSS-TM3, NIST-ISAM, OWASP MASVSS, ISSAF, and PTES. These models are briefly introduced and the most suitable model is selected according to the requirements defined previously.

OWASP MASVS is security testing standards listing based on preventing OWASP Mobile Top Ten Risks. This standard listing contains a step-by-step checklist to verify OWASP Mobile risks with additional reverse engineering resiliency standards for mobile applications as an appendix. MASVS listing is divided into two security levels where first-level MASVS-L1 defines generic security requirements and the second level MASVS-L2 includes in addition to L1 requirements additional standards for protecting against more sophisticated attacks like SSL pinning [Willemsen, *et al*, 2021]. OWASP MASVSS as itself does not have any detailed guidelines about the testing process nor tools used for testing the mobile applications. Therefore OWASP MASVS and OWASP MSTG are used as a technical references for the security testing process but not as a model [Mueller, *et al* 2020]. According to MASVS [2020], it is not a security testing model and does not include any information about running the security testing process. MASVS listing includes a set of standards for secure mobile applications. Therefore, it is not necessarily clear why OWASP MASVS has been included in this listing as security testing model.

ISSAF has different suggestions and standards for a different types of software. These standards are including realistic scenarios of attacking the different types of systems. Steps to execute in ISSAF process are Planning, Assessment, Reporting and clean-up. ISSAF methodology is not compatible with React Native security testing model as it does not have any information about testing the mobile applications [Haq and Khan, 2021].

OSSTMM includes different penetration and vulnerability testing strategies for different configurations of background systems grouped by different ways. OSSTMMs is built

according to Haq and Khan [2021] to help network developers and testers to verify and improve systems as whole security. Therefore it also is determined to be not suitable with our current study where security is tested mainly at client level and wide testing process of backend systems are determined to be out of the study's scope.

ISAM (Technology Information Security and Assurance metrics) is a set of metrics and methodologies created by National Institute of Standards and Technology. ISAM project is funded and owned by the US government and has the goal of testing and assuring the security of different systems [NIST, 2021]. The penetration testing framework by NIST consists of four phases, which are planning, discovery, attack, and reporting. However, this methodology might be outdated nowadays as there are many newer and more specific testing guides like NIST 800-163, which includes NIST organizations' current standards for testing mobile applications and vulnerabilities according NIST [Ogata, et al, 2019].

PTES (Penetration Testing Execution Standard) is a security testing model containing seven steps which are (1) Pre-engagement Interactions, (2) Intelligence Gathering, (3) Threat Modelling, (4) Vulnerability Analysis, (5) Exploitation (6) Post Exploitation (7) Reporting [PTES, 2021]. PTES is a generic penetration testing process attempting to create standardization all across security with the process including even physical office-level security. OWASP security testing processes are at the base level based on these seven steps of PTES [Haq and Khan, 2021]. PTES has also technical testing documentation available, based on these seven steps. This documentation is providing information about all-around security all the way from backend systems to mobile applications [PTES, 2021].

From these 5 sets of methodologies, there were two possible models without crucial shortages. These models were PTES and ISAM. In the end, PTES was selected as the most usable for our purposes due to its compatibility with the OWASP methodologies and its flexible nature [Haq and Khan, 2021]. Being compatible with an OWASPs model and OWASP model being built around this PTES model, will make it easier to use OWASP methodologies when testing the applications and creating the model. Parts like testing physical facilities, however, are not applicable for this study and will be ignored when model is being followed. Finally, PTES provided steps will be renamed to suit the mobile testing scenario better. The base for this new naming scheme is taken from OWASP MSTG [2020], which has a methodology for testing purely native mobile applications. High level description of this model is introduced in Figure 20.

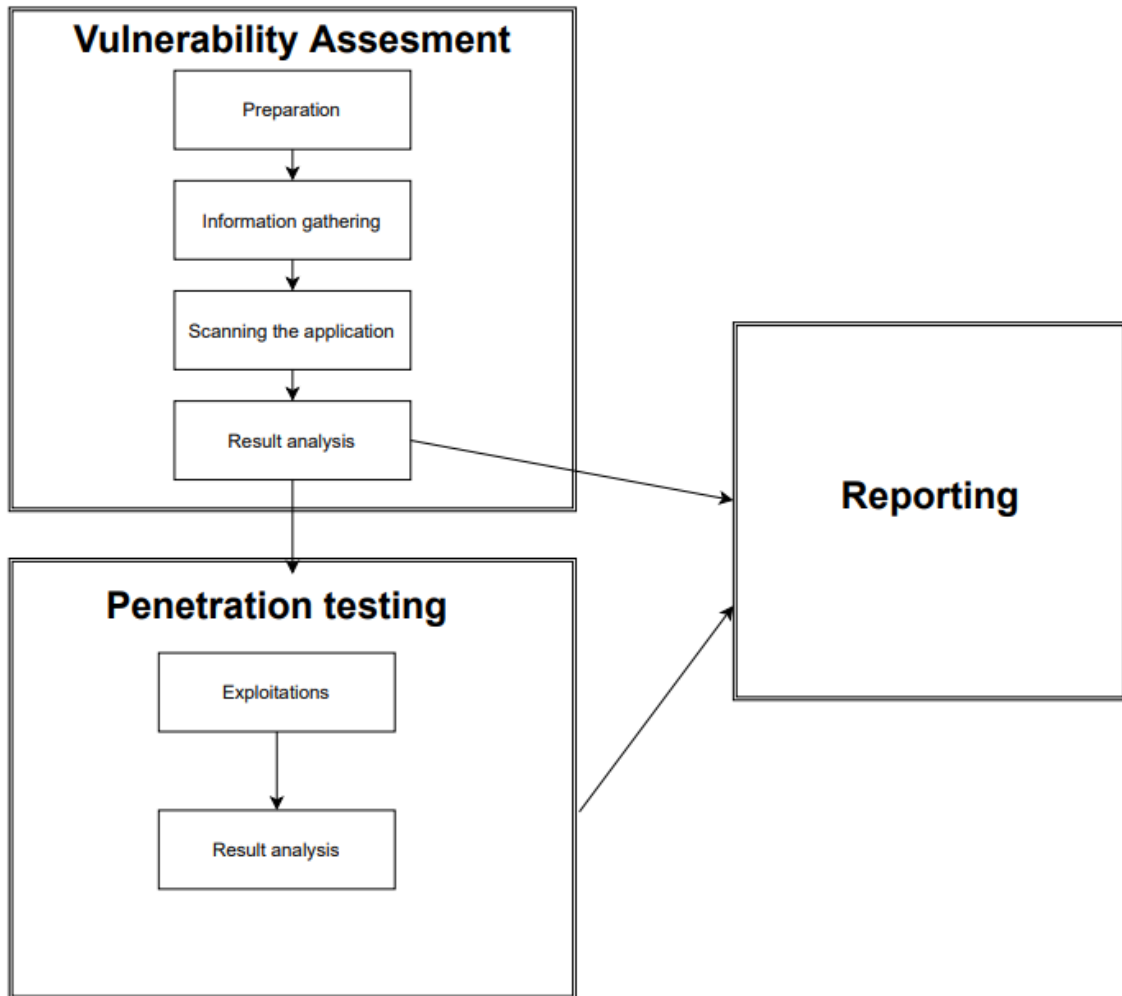


Figure 20 Mobile application security testing stages

8.1 Vulnerability assessment

The first part of the security testing process is vulnerability assessment. The main goal of the vulnerability assessment stage is to find vulnerabilities from the application by using manual, static and dynamic tools. This scanning phase is supported by preparation and information-gathering stages where information about the application is gathered by using the information provided by the customer and by executing an independent information-gathering process.

In addition to executing the scanning process and reporting found vulnerabilities to the customer, the vulnerability assessment phase is also part of the broader testing strategy where information is gathered for the second phase of the security testing process, which is the penetration testing phase. In penetration testing phase vulnerabilities and the damage they would cause to the system are validated, by exploiting the found vulnerabilities like a real world attacker would. [Shinde and Ardhapurkar, 2016]

8.1.1 Preparation phase

Vulnerability assessment starts from the preparation phase, summarized in table 4. In this stage goals and rules of the testing are discussed with the owner of the application. Here a security tester will meet up with the customer to decide on a more specific plan for the testing process.

Task	Description
Identify sensitive data	What sensitive data application stores
Define scope of testing	What is tested, information gathering level and how far to penetration test
Define timeline of testing	When testing will start and end
Request information about the application	Ask for architectural and possible other information about application
Request different builds	Gain access for different builds of application
Determine information gathering level	Discuss with the customer how much effort is used in information gathering process

Table 4 Steps in Preparation phase of security testing

This plan includes scope, where it is decided what is going to be tested and how far testers will go in terms of exploiting the application [Sugandh 2015]. This planning stage can easily be overlooked. However, according to PTES [2021] defining the scope and timeline as exactly as possible is important for a successful testing process. This is important in situations where outside security testing is being conducted. This is since usually the scope of testing will grow during the testing process if the scope is not strictly defined. Another important part of supporting the actual testing process is to determine what is and what isn't sensitive data. This should be discussed with the customer as many organizations might have data classification policies that determine what is sensitive data from the organization's point of view. If there is no active data classification policy in place, categorization can be done by using lists defined by MSTG. [Mueller, *et al* 2020]

In addition, other possible information about the application should be requested from the customer at this stage. This is unless full black-box testing where applications are downloaded from the app store is being conducted. Boduch and others [2020] suggested that already in the preparation stage both debug and production builds of the application should be requested from the customer. Having these two builds allows testers to test the application without tedious to break security controls. Also verifying the found

vulnerabilities is easier when applications logs and other debug information can be accessed easily [Mueller, *et al* 2020].

8.1.2 Information gathering

After the scope and information about the application have been gathered from the customer, it is time to conduct the first independent from the customer step of the testing process. This process can be called information gathering or reconnaissance, which is summarized in Table 5.

Task	Description
Native reconnaissance using applications package and MobSF	Determine applications architecture, included libraries, components and installed files
Network reconnaissance	Determine applications Network security policy and applications used services
Passive reconnaissance	Conduct passive reconnaissance by using open source intelligence
JavaScript environment reconnaissance	Determine JavaScript packages, JavaScript version and JavaScript engine and possible other services used by application.

Table 5 Steps in information gathering phase.

Reconnaissance according to PTES [2021] can be divided into three groups depending on the effort used during the process. These effort levels are ranging from 1 to 3. Level-one type of information gathering is conducted mainly using automated tools and in level three heavy analysis is conducted and a large number of hours is used to gather an in-depth understanding of the organization behind the tested application. This level of information gathering should be determined in the preparation phase during discussions with the customer.

This process should be separated from the data received from the customer as data gathered from the customer is already known. Therefore this stage should be executed independently from the first step. This will make it possible for the tester to determine the amount and type of information that is gatherable by a possible attacker when using only an open-source intelligence and production package [Sugandh, 2015].

The information-gathering process and its goals are different, depending on the source that is used. For example, in OWASP [2020] information gathering is more of a technical

process, only consisting of automated reconnaissance parts of the intelligence gathering process and the resources used are source code and the network that the application is located in. On the other hand, studies conducted by PTES [2021] and Sugandh [2015] defines information gathering as the process for gathering information about the target organization and environments that software is located. That is added on top of applications technical reconnaissance like defined in OWASP [2020].

In this study, the focus is on the active reconnaissance process, which is closest to PTES security level 1 and process defined in OWASP [2020]. In this active reconnaissance process, the source code of the application and the application's package are used for gathering information. In addition to the applications package, different business cases, possible organization-specific internal processes, and a picture of applications architecture should be determined during this process.

Overall this reconnaissance process includes a lot of business and scope specifics and when conducting security testing, a process for testing these specific assets should be determined case by case. Therefore, a more specific approach executed in the case study section is determined during the case study section by using documentation provided by PTES [2021] and OWASP [2020].

8.1.3 Scanning the application

In this stage, an initial set of flaws possibly exploitable by the attacker are searched from the application [PTES, 2021].

Task	Description
Running statical scanners for all environments	Scan all targeted environments and JavaScript environments statically by using tools selected.
Running dynamical runners for all environments.	Scan all targeted environments and JavaScript environments dynamically by using tools selected.
OWASP MASVS listings	Fill authorization MASVS in JavaScript environment
Possible custom authentication process	If needed test custom authentication process by reviewing source code of application and using interception proxy
Test React Native storage solution	Check manually how storage solution is implemented in React Native in source code and dynamical analysis level and find possible issues.
Business logic specific testing	Test application specific business logic by using interception proxy and source code.

Table 6 Actions performed in scanning the application phase.

More precisely this means that bulk amounts of vulnerabilities are gathered from the application by using the data gathered in previous steps [Sugandh, 2015]. At the technical

level, vulnerability gathering process includes a process of running dynamic and static tools with possible additional manual steps. These tools and steps executed should be chosen according to availability of tools and according to application-specific needs. Tools used in the study were visited in the earlier chapter.

Steps conducted in scanning the application phase as a whole are introduced in Table 6. This table contains in some React Native-specific manual steps, these manual steps are targeted to areas where automated tools are not necessarily catching all possible vulnerabilities. Therefore, these steps are executed with steps described in OWASP MSTG [2020]. The scanning process should be done with similar info as in the information-gathering phase, and execution of these tools should be done with applications security controls turned on. This simulates closest to a situation with an external attacker [Mueller, *et al*, 2020]. If strong security controls like obfuscation are found from the application, that is noted and testing is continued by using a package without security controls.

8.1.4 Result analysis

Now in the previous section set of possible vulnerabilities were gathered from the application. However, results of the vulnerability assessment process, when using mostly automated tools can be problematic as there is a high possibility that results are containing a large number of false positives [Sugandh, 2015]. That is why in this stage of the process scanned results are categorized by using methodologies defined in table 7. This process is summarized in table 7 and will start by assigning CVE or CWE identifiers to all vulnerabilities found in previous steps.

Task	Description
Vulnerability categorization	Categorization of known vulnerabilities to either CWE or CVE
Remove false positives	Removing false positives from the list of vulnerabilities
Reporting to customer	The report found vulnerabilities to the customer

Table 7 Steps for Result analysis

At the beginning of this stage, CVE is attempted to be assigned to vulnerability as it gives more accurate information about the severity and possible way of testing the specific vulnerability. If CVE identifier cannot be found, CWE identifier will be assigned to vulnerabilities instead. Where CVE identifier contains information about specific vulnerability CWE includes information about more generic weakness in the system and

works as a higher-level grouping method of CVEs [Mitre, 2021]. Found Vulnerability or weakness identifiers are then used to find information about the specific vulnerability or weakness, from vulnerability databases like NVD. Another way of finding information about vulnerabilities with open-source projects or libraries is open-source project-related discussions. These discussions can be found inside GitHub's issues tab. This found information is then used to determine whether found vulnerability is a false positive or actual security problem which is addressed further in the next steps.

After results have been analyzed and vulnerabilities for further testing are selected, a set of found vulnerabilities are reported to the customer and possible scope changes are determined.

8.2 Penetration testing

Penetration testing is the second step of the security testing process. In the penetration testing phase, the security tester systematically assesses the security of the system by attempting to actively attack the target system [Shinde and Ardhapurkar, 2016]. This attack is conducted by using information and set vulnerabilities found in the previous vulnerability assessment phase. An example of the action executed in this stage would be an attack where a security tester is attempting to execute admin functionality by using credentials without proper admin privileges. Technically this exploitation could be done for example by sending an HTTP request to API requiring admin privileges without proper ones.

The goal of the penetration testing phase is to identify the validity of different risks and vulnerabilities found in the previous sections. This information about exploitations results gives a tester and a target organization a picture of the overall state of security inside the target system than vulnerability assessment would be by itself. This information can include, for example, information about technical and business impacts of the vulnerability if actual exploitation would happen in a production environment

In the context of the security testing process, penetration testing will be executed after the vulnerability assessment phase where the tester uses vulnerability assessment tools to identify the possible vulnerabilities which will be exploited in this stage. [Palacios, *et al*, 2019]

8.2.1 Exploitations

On the process level, exploitations should be executed after a well-performed vulnerability assessment phase, where a list of prioritized vulnerabilities inside highly valuable assets is established [PTES, 2021]. This exploitation process is represented in Table 8 and is started by gathering possible exploitations methods for all vulnerabilities found in previous stages. When this is done to each vulnerability separately, an output is the customized set of exploits for all vulnerabilities, taking into consideration both a use

case of the exploitable part of the system and the overall technical implementation of the system. This tailored exploitation process is crucial for a successful penetration testing process [PTES, 2021]. A three-step approach identifying, customizing, and executing the exploitations is the reason why the penetration testing process is usually a far more laborious process than vulnerability assessment, which can cause its execution to be often neglected [Shinde and Ardhapurkar, 2016; Mueller, et al, 2020].

As the JavaScript environment is similar in iOS and Android environments, JavaScript-related vulnerabilities have to be only exploited in a single environment. There are no specific rules for selecting, which environment is used for exploiting JavaScript vulnerabilities. Selection can be made by choosing an environment that has the least amount of native vulnerabilities found in it or it can be done depending on the availability of testing devices.

Now everything should be ready for the actual exploitation process. In the exploitation phase, according to Mueller and Others [2020], flexibility is important and if problems occur or exploitation did not succeed, it is usually advisable to change the exploitation approach and adapt by modifying exploitations.

Task	Description
Environment for JavaScript	Select environment for exploiting JavaScript vulnerabilities.
Prepare attacks	Prepare and select exploits for attacking against selected vulnerabilities.
Remove vulnerabilities without exploit	Remove vulnerabilities without the exploit from the process.
Execute exploits and gather information.	Execute selected exploits and gather information about exploits for result analysis.

Table 8 Steps for Exploitation phase of penetration testing

8.2.2 Result analysis

At this stage, the technical part of the testing process has been completed and the tester has a list of exploited vulnerabilities and results of exploitations. In the beginning, a security tester removes false-positive vulnerabilities detected in the exploitation phase. In this stage, it is good to acknowledge that if exploitation cannot be found or it failed, that does not necessarily mean that exploitation of that vulnerability is impossible. Steps executed during the result analysis phase are introduced in table 9.

After false positives are removed, the security tester will calculate CVSS scores for the false-positive cleaned vulnerability set. For vulnerabilities that are already previously disclosed, an established CVSS score is assigned according to the CVE identifier. This scoring information can be used by the customer to prioritize the fixing process. More information about the CVSS scoring system can be found in chapter 7 where vulnerability categorization is introduced.

Task	Description
Remove false positives	Remove false positives established in the penetration testing phase.
Generate CVSS scores	Assign CVSS scores to all vulnerabilities, by either calculating it or using existing CVSS scores of disclosed vulnerabilities.

Table 9 Steps for result analysis phase of penetration testing

8.3 Reporting

In this final phase of the process, findings and information about testing conducted will be communicated to the customer. This information should include according to PTES [2021] assets tested, the scope of testing, comprehensiveness of the testing process, and methods used during the testing process.

8.4 Different platforms and model

At the technical level vulnerability analysis and penetration testing of mobile applications should be conducted separately to all targeted platforms which in the context of this study are Android and iOS [Ogata, et al, 2019; Mueller, *et al* 2020]. In this section general non-technical model of security testing is formed. This model is used for the security testing process executed in the next chapter. Therefore both platforms and the hybrid framework are included in this model. All parts if not informed otherwise should be conducted to each targeted platform with platform-specific tools, considering application specifics.

9 Case study security testing React Native application

In this chapter, tools and the security testing model defined in previous sections will be used to conduct security testing of the real-world React Native application. This application has a wide user base and it's being used by thousands of users every month. The application targets both major mobile platforms iOS and Android and contains platform-agnostic JavaScript parts. This hybrid application tested is in the category of medical applications, which means that medical data is displayed to the end customer by the application. In addition to sensitive data displayed, the application has functionality for discussing with medical professionals. At the end of the discussion payment of the service happens through the application. Due to these different categories of sensitive datasets displayed and generated by the application, it is important for the application to have a sufficient level of protection around its transmitted and stored data. This is the reason why the application was found to be sufficient for the purposes of this study.

The case study is started as defined in the testing model, by collecting a set of vulnerabilities by vulnerability assessment. After the collection of vulnerabilities, penetration testing described will be conducted to validate these found vulnerabilities. At the end of the process, found vulnerabilities and issues are reported to the customer.

9.1 Preparation phase

In the preparation phase, communication with the customer is established, specifics about testing are discussed, and possible information about the application is requested.

In the first phase of vulnerability assessment, sensitive data that the application stores and uses is defined. The application tested consists of many different categories of data that can be defined as sensitive. This sensitive data in the application includes personal details about a user (Email, postal address, social security number, coarse location, etc), medical records, and discussions history with the medical professionals. In addition to this sensitive data about the user, the application consists of technically sensitive data for example, long-term authentication tokens. This data as a whole will be searched from the application during the study.

The second part of the preparation phase is to request information about the system and the application. This system as a whole involves the mobile application, API service exposed to a public network and backend services in the internal network consisting of sensitive data. In addition to backend systems, there is also a user interface used by medical professionals to discuss with the customer.

The third part of the preparation phase is to define the scope for the testing. Data gathering is conducted without OISINT and by using minimal PTES L1-level data gathering. The testing process will be done as white box testing, with full knowledge of the target system, including full access to source code. It was stated in the previous chapter that the system overall consists in addition of mobile application, multiple

backend services, and an additional user interface. However, as this study is based on security testing methodologies of React Native mobile applications, the scope of the testing is set strictly to include only application and the direct APIs that are usable through the application. No network scanning nor backend service-related testing is conducted during this study. However, as stated in studies conducted by PTES [2021] and Muller and others [2020], it would be advisable to validate back-end systems as well if a comprehensive security testing process of mobile application would be conducted.

In technical sense scope of the testing process will consist of executing vulnerability assessment using tools described in chapter 6. In addition to executing these tools, OWASP MASVS checklists described in chapter 5 and React Native specific vulnerabilities introduced in chapter 4 will be tested. After the vulnerability assessment process has been completed and false positives have been removed, the found vulnerabilities will be moved to the penetration testing phase, where exploitations for found vulnerabilities are studied, and vulnerabilities with found exploitations are attempted to be exploited. In the end, false positives found in the penetration testing phase are removed and exploited vulnerabilities will be categorized by CVSS scores. This information at the end is reported to the customer. This first stage of testing is summarized in Table 10.

Task	Summary
Identify sensitive data	Identified and described in chapter
Define the scope of testing	Testing conducted in a Staging environment. Strict scope is defined in the main body of text.
Define timeline of testing	Duration of testing process is three weeks. One for penetration testing and one for vulnerability assessments
Request information about the application	Information requested—testing conducted as Whitebox and process has free access to application and backend systems.
Request different builds	Different builds and source code accrued

Table 10 Summary of the preparation phase

9.2 Information gathering

In this stage, information will be gathered independently from the customer, marking a start to the independent testing process. Results of this process are introduced in Table 11. The information-gathering process is executed to native environments and the JavaScript environment of the application. A more comprehensive testing process should

contain its own stages for open-source intelligence and network scanning. Nevertheless, as this study is based on the security testing React Native applications and conducting a full OISINT process would be a tedious process. OISINT process is determined to be out of this study's scope. However, as stated in many sources including PTES [2020] and Haq and Khan [2021] it would be advisable to conduct these OISINT steps when the testing is targeting the system as a whole.

This information-gathering process will be conducted two times once with iOS and once with Android application. In both platforms, testing is started with packages like those published in the app stores, including all security controls. If security controls of the package are strong, package without the security controls will be used to continue the testing process. In the end, if all data cannot be gathered by using application packages, source code will be used. This is done due to fact that according to Mueller and Others [2020] security controls like obfuscation are not sufficient protection methods by itself and should only be used only to make possible attacks more time-consuming and more difficult to execute. Not as only a mean of protection against attacks.

Task	Summary
Native reconnaissance using applications package and MobSF	Native environment-specific architectures and services determined. No binary protection in Android, Apple provided protection on iOS
Reconnaissance using network	Out of decided scope. More useful when testing system-level security.
Passive reconnaissance	Out of decided scope. Information gathering level decided to be L1.
JavaScript environment reconnaissance	JavaScript environment architecture and services determined. No binary protection in JavaScript bundle.

Table 11 Summary for information gathering phase.

9.2.1 Android

Initially, testing Android environment is started with the APK similar that can be downloaded from Play Store. In this APK, debug functionality is turned off, and the building process is done similarly to production packages, with possible security controls turned on. This APK is then reverse engineered by using APKTool. This APKTool

reversed package is then used to conduct JavaScript environment and native environment-specific reconnaissance processes.

The testing process is initially conducted in the JavaScript environment and is started by reviewing APK packages JavaScript files and resources. During this review, it was found that files of the Android package are readable, and no obfuscation or any other security controls are found. After the APKTool reversed package was reviewed, five different categories of JavaScript environment-related files were identified. These files are JavaScript Bundle file, I18n translations including key translation pairs, animation files, icon files, and package.json files of external dependencies. The most interesting part of the JavaScript environment is the JavaScript bundle, which contains applications business logic that looks unreadable. However, bundled files can be prettified using a tool like Js-Beautify [2021], making the applications JavaScript source code easily patchable and readable like presented in Figure 23.

```
getNewIdentificationFlowStartUrl: function(t) {
  return 0 + "/patient/auth/mobile-identification-start/" + t
},
getNewIdentificationFlowStartUrlTest: function(t) {
  return 0 + "/patient/auth/mobile-identification-start/" + t + "?isTestMode=true"
},
needsReauthentication: function() {
  return (0 + "/patient/auth/session-version").then(function(t) {
    return t.reAuthenticate
  })
}
```

Figure 23 Part of prettified and reversed Android bundle.

Two interesting JavaScript environment-related services are found. These are CodePush and Redux. The application uses the *CodePush* service to update JavaScript level changes by fetching updated bundle files from CodePush servers. That makes it easier to deliver JavaScript changes without a new play store release [Microsoft, 2021]. Redux, on the other hand, is used for helping applications to keep data consistent around the application by keeping applications' state as global [Redux, 2021].

Next native environment reconnaissance of Android is conducted. That will be started similarly as with the JavaScript, with APKTool reversed APK. Results of Androids native reconnaissance were similar as in JavaScript environment. Source code files had been minified but not obfuscated. Therefore source code of the application is freely accessible. The rest of the native part of the Androids reconnaissance process uses MobSF statical scanning, which is done with a package similar to that distributed in App Store. Results of MobSF are introduced more in detail in the following chapter when results of security scanning tools are presented.

9.2.2 iOS

There are two types of iOS applications built with newer Swift and built with older Objective-C technology. This that is tested during this study is built by using newer iOS

language Swift. This makes reverse-engineering and dynamical testing process overall harder [Mueller, *et al*, 2020]. Overall iOS applications reconnaissance process will be conducted similarly as the Android applications one; however, the reverse-engineering process provided by Mueller and Others [2020], our primary technical reference, cannot be used as its iOS-based process is mainly based on reverse-engineering Objective-C applications [OWASP, 2020].

Native iOS applications built with the Swift are obfuscated by default as the applications are shipped by using IPA packages. To circumvent this obfuscation, an application would have to be snapshotted during dynamic analysis [OWASP, 2020]. This will be skipped during this study, and the rest of the iOS reconnaissance process will be done using an unobfuscated APP file.

As intended in the case of the hybrid application, the overall architecture and used services of the iOS applications are the same as with the native Android applications. So, Therefore, the rest of the information gathering process will be conducted similarly as it was done with the Android, by using MobSF. Results of this scan will be used in the following chapters when the application is further assessed.

9.3 Scanning the application

After the background information about the application is gathered, the application is analyzed for vulnerabilities.

Task	Summary
Running statical scanners for all environments	Statical scanners were executed, and possible issues were found for all environments.
Running dynamical runners for all environments.	JavaScript and Android environments dynamically tested successfully—problems faced with iOS emulator.
OWASP MASVS listings	Authorization MASVS done in JavaScript environment.
The possible custom authentication process	Authentication process tested with source code and Burp Suite. Possible issues were found.
Test React Native storage solution	Redux state not persisted, issue found regarding Keychain and MASVS standards.
Business logic specific testing	Applications JavaScript source code briefly reviewed, and application tested using interception proxy.

Table 12 Summary for scanning the application phase

This analysis of vulnerabilities is done by using mainly different static and dynamic environment-specific security testing tools. These tools are defined in more detail in chapter 6. In addition to the usage of automated tools, some MASVS checklists and manual steps are defined during the security testing model. Those additional steps will be

used during this section to help validate areas of testing, which cannot be assessed reliably automatically. Steps executed and overall results of this stage are introduced in Table 12.

9.3.1 JavaScript environment

In the beginning, before moving to platform-specific testing processes, platform-agnostic JavaScript code will be tested. JavaScript parts of the applications are executed in both Android and iOS environments by using the same JavaScript engine. Therefore, the results of this part of testing are valid both in iOS and Android environments. The first scanner used to JavaScript environment is a statical scanner called SonarQube. SonarQube takes applications JavaScript source codes as input. Results of this scan are described in Figure 24.

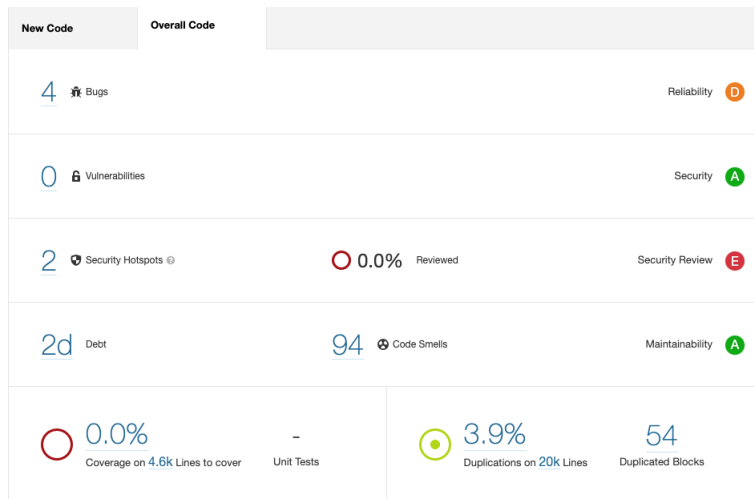


Figure 24 result of SonarQube Scan

As presented in Figure 24, four bugs, two security hotspots, and 74 code smells were found by SonarQube. Analyzing these results will be started from the Code smells. Code smells are more of an issue regarding maintenance of the project, contrary to active security issues or vulnerabilities [SonarQube, 2021]. Therefore, these code smells can be reported to the customer, but will be removed from the security testing process at this stage. After the removal of the code smells, there are still six issues left to review. From these six issues, issues with CWE or CVE tag assigned by SonarQube were selected to the follow-up stages of testing. Rest of the issues was dropped as these did not have any security issue-related information available. More detailed information about issues chosen from follow-up stages can be found in Table 13.

Category	Issue
Regex DoS	Email validator <code> /^[^\\s@]+@[^\\s@]+\\. [^\\s@]+\$/</code>
Usage of HTTP protocol	With webview origin whitelisting <code>originWhitelist={['https://*', 'http://localhost*', 'http://10.0.2.2*']}</code>

Table 13 results of SonarQube Scanning.

Now JavaScript source code has been scanned statically. The next step of the process is to scan applications' JavaScript-level dependencies for known vulnerabilities. This testing is conducted by using three different tools Yarn Audit, GitHub security advisors and OWASP Dependency Checker. Like expected all three different tools had similar results and were warning about the same three vulnerabilities. Methodologies for Finding known vulnerabilities are similar with all three of these tools, like stated in tooling section. The only difference found from outputs of these tools were that yarn audit provides more information in situations where single dependency is sub dependency of multiple libraries. In this situation Yarn audit gives multiple warnings with only single unique vulnerability found. Overall results of this stage are presented in Table 14.

Category	Issue
Possible sandbox escape	vm2 sub dependency
Regex DoS	ansi-regex sub dependency
Type confusion	Set-value sub dependency

Table 14 combined results of GitHub Security advisors and Yarn Audit.

Next first manual step of process is done. In this step authentication scheme of the application is tested. As stated earlier, authentication schemes of the applications can be highly customized, and therefore testing authentication schemes will have additional manual step added in the model. This manual step uses OWASP MASVS provided by Willemsen and others [2021].

Half of the MASVS checklists entries are marked as N/A, which will mean that these parts of the MASVS were not tested. These rows were eliminated either because the system did not have that functionality implemented at all or the issue was regarding only back-end systems and was deemed to be out of the scope. There was a total of three issues found; row 4.1 is due to possible vulnerability, which will be introduced later where a follow-up login process is enforced only at the application level. Two final issues, 4.4 and 4.10, are functionalities that have not been implemented in the application. These two issues are more harmful due to possible issues introduced later where applications' long-term authentication token is compromised on the device level. There is no way of ending the compromised session nor any additional protection layers around sensitive transactions, therefore if authentication token is compromised, user should be able to end the session. Results of MASVS listing can be found in Figure 25.

4.1	MSTG-AUTH-1	If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.	✓	✓	Fail
4.2	MSTG-AUTH-2	If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user's credentials.	✓	✓	N/A
4.3	MSTG-AUTH-3	If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm.	✓	✓	N/A
4.4	MSTG-AUTH-4	The remote endpoint terminates the existing session when the user logs out.			Fail
4.5	MSTG-AUTH-5	A password policy exists and is enforced at the remote endpoint.	✓	✓	N/A
4.6	MSTG-AUTH-6	The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.	✓	✓	N/A
4.7	MSTG-AUTH-7	Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire.	✓	✓	N/A
4.8	MSTG-AUTH-8	Biometric authentication, if any, is not event-bound (i.e. using an API that simply returns "true" or "false"). Instead, it is based on unlocking the keychain/keystore.		✓	N/A
4.9	MSTG-AUTH-9	A second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced.		✓	Pass
4.10	MSTG-AUTH-10	Sensitive transactions require step-up authentication.		✓	Fail
4.11	MSTG-AUTH-11	The app informs the user of all sensitive activities with their account. Users are able to view a list of devices, view contextual information (IP address, location, etc.), and to block specific devices.		✓	Fail
4.12	MSTG-AUTH-12	Authorization models should be defined and enforced at the remote endpoint.	✓	✓	Fail

Figure 25 Authentication scheme MASVS

The next step of the JavaScript level testing process is to review the way the storage solution is implemented at React Native level. The review will be done using the application's source code. The application uses AsyncStorage and react-native-keychain (<https://github.com/oblador/react-native-keychain>) to persist data over applications restarts. Starting with AsyncStorage. The application uses AsyncStorage for storing data globally over application restarts in an unencrypted fashion. AsyncStorage is implemented differently in different environments, on Android local database SQLite is used and in iOS separate files [Facebook D, 2021]. AsyncStorage is used by the application to store only insensitive data such as the language information of a user. Therefore, there are no issues found regarding storing data to AsyncStorage. Next React-native-keychain is reviewed. React-Native-Keychain is used by the application to store authentication tokens. This usage of react-native-keychain by itself does not create any vulnerabilities to the application. However, there is risk involved with usage of react-native-keychain, which is introduced more in detail in chapter 4, where it is stated that according to Willemsen and others [2021], storing sensitive data to an iOS keychain without initial

encryption breaks data storage standards. This will create a standard breach for the iOS environment.

The final storage technology to be reviewed is redux. Redux is used to persist applications state globally in runtime, and in the application, this state contains sensitive data, which, if persisted over application restart, would pose vulnerability. However, the Redux state is not persisted over application restarts, making it a suitable solution [Facebook, 2019].

The next part of the JavaScript environment-specific testing process is to review the business logic and functionalities of the application. This testing is conducted by using the application in its runtime state and interception proxy Burp Suite. Process using the interception proxy will allow us to see how the application interacts with backend systems.

After, the application's functionality was reviewed it was found that the application uses safe communication methods all-around to interact with different backend services. However, two issues were found regarding applications authentication scheme. Firstly when a user initially logs in to the application, this happens securely. But during the second time when a user logs in by using the same device, the application only verifies login credentials in the front-end and decrypts the password before the user is authenticated, making the application's authentication process vulnerable to code tampering attacks. This attack can cause possible circumvention of the authentication process or leakage of customers' device-level password. Vulnerability is possibly exploitable by malware to target larger groups of users. Final issue found was possible circumvention of the payment process. It was found that either by code tampering or by sending a forged WebSocket message, a user is possibly able to circumvent the payment process by sending a WebSocket message ending the chat before the invoice has been made to a customer. Results of this review of the application's business logic can be seen in Table 16.

Category	Issue
Insecure authentication	Possible circumvention of the application's login process.
Insecure authorization	Possible circumvention of the payment process.
Insecure data storage	Possible leakage of user's application-level password

Table 16 Issues found from applications business logic

9.3.2 Android

In this chapter, the platform-specific testing will be conducted by searching vulnerabilities from React Native-built Android application. This testing is executed similarly as it was conducted to a JavaScript environment by using security testing tools,

MAVS checklists, and manual steps defined previously in the study. The environment used for the testing process includes Genymotion emulator, Android debug bridge command-line tools and emulated Android device with SDK version of 29.

The testing process is started by statically and dynamically scanning the application by using MobSF. Results of this scan can be found in Table 17. Overall, MobSF gave a lot of info, warning, and vulnerability level information. However, almost all of the information found by the tool was found either in native code of React Native’s core or external dependencies. In this study, libraries are validated in Android environment by scanning them for known vulnerabilities later in this chapter. Therefore, only a small part of the most prevalent issues found by MobSF are included in Table 17. When initial false-positive assessment was conducted after MobSf scan, it was found that there is a high possibility of false positives to be present among these issues found by MobSf.

Category	Issue
Cleartext Storage of Sensitive Information	MobSF Specifies seven possible files
Use of Insufficiently Random Values	Tool specifies three possible files
Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking	Usage of encryption mode CBC
Use of a Broken or Risky Cryptographic Algorithm	One file uses SHA1 and one MD5 hashing.
Exposed Dangerous Method or Function	Insecure configuration of webview
Application Data can be Backed up	Data can be extracted when USB debugging is enabled
Unprotected activities	Possibly harmful unprotected activities
Cleartext Storage of Sensitive Information	Possible WebView cookie information in Sqlite

Table 17 Result of statical and dynamical analysis by MobSF

Next Android application is tested with is QARK. Overall cleaned results of this scan can be seen in Table 18. Similarly, as in the previous chapter with MobSf, this scan gave hundreds of different warnings about possible vulnerabilities in logging, WebView configurations, and unprotected activities. Similarly, as with MobSF majority of issues referencing to external dependencies or React Native core files were dropped already at this stage of testing.

QARK found some of the issues that were found already by MobSF. These issues were also left outside of Table 18. Also, many of the problems regarding the configuration of the WebViews are only prevalent in very early Android versions, which are not targeted by the application.

Category	Issue
Logging in a production application	Production application possibly emits logs
Insecure WebView configurations	Multiple places with insecure webView configuration

Table 18 Results of QARK scan.

Next the Android application’s Java code is scanned using SonarQube, which will use source code contrary to other Android environments tools. Results of the scan are described in Table 19.

Category	Issue
Bad qode Quality	The method will be continued after a thread is interrupted
Unprotected component	The component is exported and not protected

Table 19 SonarQube results for Android

The final part of the native Android vulnerability assessment is to scan codebases Android-related dependencies for the disclosed vulnerabilities. This scanning is done by using OWASP Gradle dependency checker. Results of this scan were finding 30 vulnerabilities found from Androids' native dependencies. However, validating ten external dependencies, including thirty known vulnerabilities, would take a considerable amount of time without giving any actual benefit for our React Native-related study. Therefore, these vulnerabilities are not validated in the penetration testing stage. However, these thirty vulnerabilities are reported to the customer as they were found.

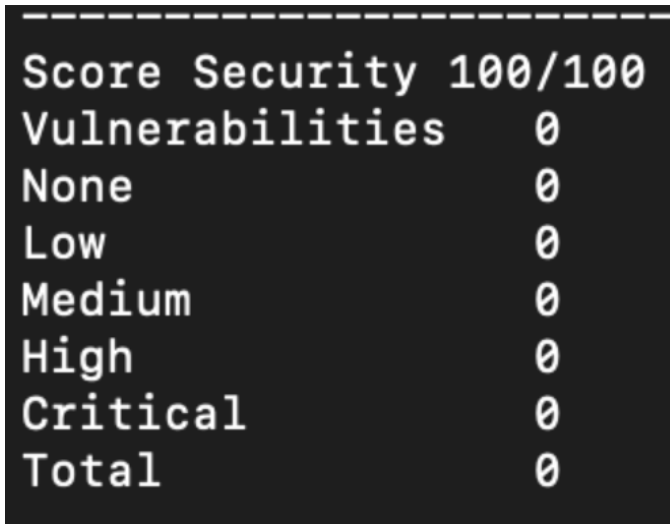
9.3.3 iOS

In previous sections, platform-agnostic JavaScript and Android environments were vulnerability assessed. In this chapter, the last remaining environment, iOS, is vulnerability assessed. The assessment process is started by analyzing the applications package statically by using MobSF. This statical scan of the application did not reveal any vulnerabilities from the application, as described in Figure 26.



Figure 26 MobSF result for iOS statical analysis

Next, the application’s Swift source code is statically analyzed by using the tool Insiders. Insiders' statical scan also gave similar results as MobSF, with a security score of 100 without any vulnerabilities found. This result of Insiders is illustrated in Figure 27.



```
Score Security 100/100
Vulnerabilities 0
None 0
Low 0
Medium 0
High 0
Critical 0
Total 0
```

Figure 27 result of Insiders statical analysis of iOS application

Next, OWASP dependency checker is used to assess iOS applications dependencies. The tool found from the iOS environment's dependencies 14 vulnerabilities included in 3 dependencies. However, the tool gave vulnerabilities a low confidence score, which means that vulnerabilities will have a high probability of appearing to be false positives. As decided in the Android environment, penetration testing this number of vulnerabilities in the native environment was determined to be outside of the study's scope. Therefore, these vulnerabilities will be reported to the customer as they are found and will be dropped from the iOS testing process.

Finally, the iOS application was analyzed dynamically by using Frida and commonly used snippets found in Frida A [2021]. However, like it was stated previously, statical analysis of iOS will be conducted by using the iOS simulator. The study uses an iOS simulator because there was no possibility of getting access to a rooted iOS device during this study. In the Android environment, emulators are working similarly to real devices. However, in a simulated environment of iOS, bytecodes are compiled to X86, which is a different architecture than is used in real physical devices. This difference will cause the testing environment to differ from the actual iOS environment. Testing will not have similar functionality to an actual device; therefore, it is advised to use an actual device [OWASP, 2020].

9.4 Result analysis

In this section, vulnerabilities gathered in the previous chapter are categorized according to vulnerability categorization methodologies introduced in chapter 7. After the categorization process, false positives are identified, and found false positives are removed from the vulnerability tables. After false positives are removed, the remaining vulnerabilities will be reported to the customer and are used as a base for a penetration testing phase where these vulnerabilities are validated. Steps executed and descriptions of results are presented in Table 20.

10 Task	Summary
Vulnerability categorization	All vulnerabilities categorized where CWE or CVE identifier was found
Remove false positives	False-positives removed by researching found vulnerabilities.
Reporting to customer	Initial platform-specific listings without penetration testing were reported to the customer.

Table 20 Summary for the result analysis phase

9.4.1 JavaScript

In Table 21, all JavaScript environments issues were mapped to CVE or CWE identifiers. There are vulnerabilities where neither CWE nor CVE identifiers cannot be found for vulnerability. These issues where any identifier cannot be found were specific to the application's functionality.

Category	Issue	CVE/CWE	Severity	False-positive
Regex DoS	Email validator / [^] [[^] \s@]+@[[^] \s@]+\.[[^] \s@]+\$/	No CVE custom regex	Medium-high	No
Usage of the HTTP protocol	With webview origin whitelisting originWhitelist={['https://*', 'http://localhost*', 'http://10.0.2.2*']}	CVE-2019-6169	7,5	Yes
Possible sandbox escape	vm2 sub dependency	CVE-2019-10761	8,3	No
Regex DoS	ansi-regex sub dependency	CVE-2021-3807	7,5	Yes
Type confusion	Set-value sub dependency	CVE-2021-23440	8,55	No
Insecure authentication	Possible circumvention of the authentication process	Custom no CWE	Custom	No
Insecure storage	Usage of iOS keychain to store sensitive data	Standard breach	No severity found	No
Insecure authentication	Possible circumvention of payment process	Custom no CWE	Custom	No
Insecure Data Storage	Possible leakage of user's application-level password.	Custom no CWE	Custom	No

Table 21 false-positive cleared and categorized JavaScript vulnerabilities

Now, this CWE and CVE information about vulnerabilities will be used to remove false positives. There was a total of 2 false positives identified during the process. These

identified false positives are vulnerabilities CVE-2019-6169 and CVE-2021-3807. Firstly vulnerability CVE 2019-6169 was identified as false-positive. That was because the tool, which found the vulnerability, advised that if an insecure HTTP protocol was only used with the local environment, it does not create risk for the production environment. The second vulnerability CVE-2021-3807 was found inside the application's dependencies. This dependency Ansi-regex is the transitive dependency located in-depth of four, the overall transitional dependency tree of ansi-regex is eslint > table > string-width > strip-ansi > ansi-regex. As it can be seen, the ansi-regex was initially included in the application by eslint, and eslint is used only during the development to enforce coding conventions. Therefore, this issue is not prevalent in the production environment.

9.4.2 Android

Now a similar process like was conducted to JavaScript vulnerabilities is conducted in the Android environment. The final results of the mapping can be seen in Table 22. Vulnerability listing in Android environment contained seven false positives, the first is CWE-312 cleartext storage of sensitive information to files. These seven files are including in the example CodePushDeploymentKey, which is meant to be embedded in the source code [Microsoft, 2021]. The second type of data MobSF categorized as possibly sensitive is the state key of connected headphones in Android device. This was also determined to be a false positive.

The following three false positives CWE-330, CWE-649 and CWE-327 were all regarding insecure cryptographic methodologies. However, when further reviewed, these insecure cryptographic issues were all referencing to either a situation where the library imported to the application did implement MD5 functionality, but the application was not using it. In the second scenario applications, dependencies did use insecure means of cryptography. Still, these were only used to generate keys for non-sensitive functionalities such as generating an identifier for push notification tokens.

Category	Issue	CVE/CWE	Severity	False-positive
Cleartext Storage of Sensitive Information	MobSF Specifies seven files	CWE-312	7,40	Yes
Use of Insufficiently Random Values	Tool specifies three possible files	CWE-330	7,50	Yes
Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking	Usage of encryption mode CBC	CWE-649	7,40	Yes
Use of a Broken or Risky Cryptographic Algorithm	One file uses SHA1 and one MD5 hashing.	CWE-327	5,90	Yes
Exposed Dangerous Method or Function	Insecure configuration of webview	CVE-2019-10761	8,30	No
Application Data can be Backed up	Data can be extracted when USB debugging is enabled	CVE-2017-16835	7,50	No
Unprotected components	Causes settings activity and other activities being shared with other applications	CWE-926	No severity found	No
Cleartext Storage of Sensitive Information	Possible WebView cookie information in Sqlite	CWE-315	No severity found	No
Logging in production application	Production application possibly emits logs	CWE-532	No severity found	No
Bad code Quality	Method will be continued after thread is interrupted	CWE-215	Low	Yes

Table 22 Possible Android vulnerabilities categorized

9.4.3. iOS

Only possible vulnerabilities found in the iOS testing phase were regarded as out of this study’s scope. So, iOS result analysis will be skipped, and limitations of testing established in previous sections will be reported to the customer.

9.5 Exploitations

As there were a lot of possible vulnerabilities found in scanning the application phase, the exploitation phase is divided into two sections: Android and JavaScript. There is no iOS section in the penetration testing phase as no vulnerabilities were found inside that environment.

The first step of the process will be to define the environment that JavaScript is tested on. This step will be executed only to the JavaScript environment. The rest of the steps will be executed in both Android and JavaScript environments. Results and description of the execution of the exploitation phase can be seen in Table 23.

Task	Summary
Environment for JavaScript environment testing	iOS environment selected for testing JavaScript vulnerabilities.
Prepare attacks	Attacks were prepared and possible exploitations were found for Android and iOS environments
Remove vulnerabilities without exploitations from this stage	Some vulnerabilities like transitional dependencies did not have exploitation found.
Execute exploits and gather information.	All planned exploitations were executed successfully.

Table 23 Summary of the exploitation phase

9.5.1 JavaScript

The Exploitation process is started from penetration testing JavaScript vulnerabilities. These vulnerabilities are initially attempted to be exploited in the iOS platform, as there were no exploits found in the iOS environment. This aims to make the penetration testing process more comprehensive in terms of platforms used. In case of testing is not possible due to iOS simulators limitations, described earlier. Android environment is used for testing the vulnerability.

9.5.1.1 Preparing the attacks

The testing process will be started by preparing the attacks. This is conducted by researching information about found vulnerability with either CVE or CWE identifier. If this does not provide enough information, a general Google search about the exploitation method is conducted. The first attack built is Regex Dos without an assigned CWE or CVE identifier. ReDos is possible in situations where grouping is done with repetition with additional repetition inside like (a+)+ or by having similar alternation inside grouping, which is repeated, example being (a|aa)+ [Weidman, 2021]. In our case the possible vulnerable regex is `/^[^s@]+@[^s@]+\.[^s@]+$/` and it is exploited by using the strings with repeating @ and . characters.

Next two vulnerabilities CVE-2019-10761 and CVE-2021-23440 are found inside dependencies of the project. These dependencies are sub-dependencies of React Native framework itself and React-Native-Codepush, according to Yarn Audit scan. Usage of these libraries was studied by finding where these libraries were used by using the source code. There were no implications found that the application would be using vulnerable functionalities of these sub dependencies. Therefore, there are no possible penetration methods found and these will be dropped from the penetration testing

process. However, as research of these vulnerabilities consisted in addition of applications source code multiple open-source codebases, there is a possibility that these vulnerabilities are still possibly exploitable and updating version numbers of these dependencies is most likely a relatively short process, fixing these issues is recommended to the customer.

The next possible vulnerability inside the JavaScript environment is the possible circumvention of the follow-up authentication process. Exploitation of this vulnerability is done by using code tampering. Code tampering is executed after initial safe login has been successfully accomplished. After initial login, the application will be closed, persisting the session inside the application's safe storage. After this initial login process, the application is tampered with. This tampering process has the aim to allow the pin-code screen to be bypassed. This second login process is only enforced on the application level, and there is no credential validation inside the backend.

Next vulnerability exploited is the possible circumvention of the application's payment process. This is attempted to be exploited by closing the chat before the invoicing has been completed. This is done by sending a forged WebSocket message to the backend service. Second possible way of bypassing the payment process would be by using code tampering, which would reveal a button for closing the chat, that should be invisible.

The final attack built for the JavaScript environment is the possible leakage of the pin code that the user uses before the application successfully authenticates the user. This vulnerability is attempted to be exploited with the application in a similar state as in, bypassing the authentication process vulnerability. That is attempted to be exploited by code-tampering the applications login process and attempting to view pincode inside an alert dialog or by using the applications logs.

9.5.1.2 Exploiting the vulnerabilities

In this stage, all attack methods against these possible vulnerabilities are researched in JavaScript environments. Next, attacking against these vulnerabilities will be started. First vulnerability exploited is ReDos against email validation regex. This was attempted to be exploited by using different valid and invalid emails with different repeating characters, and exploitation failed. Therefore, this vulnerability is deemed to be false-positive.

The next vulnerability exploited is the possible circumvention of the authentication process by using code tampering. This was tested by tampering the JavaScript code of application in runtime. The result of the tampering was that the login process was successfully bypassed without the correct pin code. This was done by removing one condition in the if-sentence inside the login process, which compared the original password against the input.

The following vulnerability exploited is the possible circumvention of the payment process. This can be tested by either sending a fully forged WebSocket message to end the current chat or by removing the conditional structure hiding the button, which has functionality for ending the chat. Results of this exploitation were that vulnerability was successfully exploited by using both of the methods.

The final vulnerability attempted to be exploited is the possible leakage of applications device-level password, provided by a user. This is tested by tampering with the application's login process to include a logging statement, which logs the actual working password when it is compared to the user-entered password.

The final issue in the application, which is not a vulnerability but a standard breach. This will not be tested by penetration testing, but as stated in chapter 4, customer is recommended to encrypt data before inserting it into the iOS keychain to prevent other applications from sniffing unlocked iOS keychain data.

9.5.2 Android

In this section the same process as with JavaScript environment will be done to Android environment with an Android-specific listing of possible vulnerabilities found in it.

9.5.2.1 Preparing the attacks

Preparing attacks in Android environment is started with the possible vulnerability CVE-2017-16835. This vulnerability possibly allows leakage of sensitive data through creating Android Backup. Vulnerability is exploited by starting the application and going through the application's functionalities to generate data for the application. After data has been generated and the application is backed up, and those backups are decompressed, this data will be read and analyzed by using steps defined by OWASP MSTG [2020].

The next vulnerability exploited is unprotected activities. This will be done by using Drozer, which allows executing actions like malicious applications would. All activities are attempted to be executed without the initial login process as an anonymous user. The second step is to perform these activities with proper user credentials inserted into the application with the application background.

The third vulnerability being exploited is the cleartext storage of sensitive information. In this exploit application SQLite database is being reviewed, and a possible long-term authentication token is attempted to be extracted. If extraction will succeed call to backend service is attempted to be made with the token to verify it.

The fourth exploitation attempted to be exploited is CVE-2019-10761. In this exploitation WebView possibly exposes insecure functionality. QARK reports multiple similar possible vulnerabilities. All of these vulnerabilities have proof-of-concept exploitations provided by QARK, which will be attempted to be executed.

The fifth exploitation attempted is to find what information is retrievable from the applications logs. This is done by using the tool Android debug bridge to fetch device log files. However, as stated by Mueller and others [2020], logging overall in a production application is bad practice and gives attackers information about the target application. Therefore, if the application is deemed to emit logs, it can be seen as a weakness regardless of the sensitivity of these logs.

9.5.2.2 Exploiting the vulnerabilities

The Android exploitation process is started by attempting to exploit the vulnerability CVE-2017-16835 and testing the application's backup process. This is done by using the backup tool provided by ADB. Results were that the Application can be backed up successfully and applications backup files by themselves do not contain any sensitive information about the user. However, these backup files include long-duration cookie which is leaked from WebView. This testing also confirmed vulnerability CWE-315 where the persistent token is readable in plaintext on the files. Next, this token found will be verified by sending forged HTTP requests requiring authorization and using this token. Sending this HTTP request succeeded, which validated that token is valid. Therefore, in addition to vulnerability CVE-2017-16835 regarding backing up the applications, vulnerability CWE-315 storing sensitive data as plaintext was also validated.

The next exploitation is attempting to execute activities exported by the Android application by using the tool Drozer. The result of these activities attempted to be executed the application in a background state with proper login credentials entered were that activities exported did not break the application's authentication process nor leak sensitive data. Therefore there are no vulnerabilities found regarding unprotected activities. However, customer can be informed that according to OWASP [2020], exporting activities without proper need is not recommended and is very sensitive for vulnerabilities to arise.

The final exploitation to be attempted to exploited is CVE-2019-10761, where WebView has according to QARK different possible vulnerabilities in its configurations. These issues were attempted to be exploited, but these exploitation files, that were referenced by the tool were not found inside QARK exploitation APK nor anywhere from QARKs source codes. Unanswered Issues from 2019 were found from QARK [2019].

9.5.3 iOS

As stated in the result analysis phase of vulnerability assessment, there were no iOS-specific vulnerabilities that needed to be penetration tested found. Therefore, this section of penetration testing iOS will be skipped as well.

9.6 Result Analysis

In this stage of the testing, all vulnerabilities have been attempted to be exploited and false positives have been removed from the vulnerability listing. So overall rest of the vulnerabilities, weaknesses and other issues found will be reported to the customer. Before reporting these issues to the customer, a final listing for all validated vulnerabilities and weaknesses will be composed, and severity scores will be assigned to all vulnerabilities. Final vulnerabilities with severity and CVE identifiers added to them are introduced in tables 25, 26 and 27.

Category	Issue	CVE/CWE	Severity
Possible sandbox escape	vm2 sub dependency	CVE-2019-10761	8,3 - High
Type confusion	Set-value sub dependency	CVE-2021-23440	8,55 - High
Insecure authentication	Possible circumvention of authentication process	Custom no CWE	5.3 – Medium
Insecure authentication	Possible circumvention of payment process	Custom no CWE	7.1 - High
Insecure authentication	Removing active session by logging out is not possible	Standard breach	-
Reverse Engineering	Application does not implement any protection against Reverse-engineering	Standard breach	-
Insecure storage	Usage of iOS keychain to store sensitive data	Standard breach	-

Table 25 Summary of final issues in JavaScript Environment

Category	Issue	CVE/CWE	Severity
Exposed Dangerous Method or Function	Insecure configuration of webview	CVE-2019-10761	8,30 - High
Application Data can be Backed up	Data can be extracted when USB debugging is enabled	CVE-2017-16835	6.50 - Medium
Cleartext Storage of Sensitive Information	WebView cookie information in Sqlite	CWE-315	7.5 - High
Insecure dependencies	30 vulnerabilities in Native android dependencies	Multiple different	Low-High
Logging in production application	Production application possibly emits logs	Standard breach	-
Reverse engineering	Application does not implement any protection against reverse-engineering	Standard breach	-

Table 26 Summary of final issues found in Android environment

Category	Issue	CVE/CWE	Severity
Insecure dependencies	14 vulnerabilities in Native iOS dependencies	Multiple different	Low-High

Table 27 Summary of final issues found in iOS environment

10 Conclusions

A security testing model for React Native applications was created during this thesis'. In the end, this model was created by using OWASP methodologies and the PTES model as a base. That model was built on top of a research of React Native's security landscape, and its platform specifics were assessed similarly as in a study conducted by Hale and Hansson [2015].

Mueller and Others [2020] have stated that there are some platform specifics involved when testing hybrid applications. However, this thesis did not define what these platform-specifics would be. On the other hand, Wällstedts [2019] and Borja and Others [2021] executed testing processes similarly to both hybrid and pure native applications. Overall there are no standards regarding testing hybrid mobile applications. This issue of lack of standards was studied, and it was found that there are things like risks stated in chapter 4 that are prevalent in React Native environment, and some problems were found in the case study section regarding all those issues. So overall, if conclusive security testing is to be performed, methodologies provided by OWASP and security testing tools used to analyze native applications should not be the only form of vulnerability assessment, when hybrid mobile applications are security tested.

It was found that OWASP methodology and pure native vulnerability assessment tools provided by MSTG [2020] are enough for testing native parts of React Native applications. However, the problem faced with these tools when applied to React Native applications is that they are built and intended to be used with Pure Native applications. Pure native applications have a different structure and intended functionality than React Native applications. That is obvious for example in a situation where React Native application is built in a way that only single Android Native Activity represents the whole React Native application, and in Native Android applications, every screen is its own activity [Google, 2021]. Therefore, when statical analysis tools like QARK and MobSF were used to Android application built with React Native, it gave a total of hundreds of possible vulnerabilities and warnings referencing React Native core files and external dependencies. Therefore, at the current state, there are no at least open-source tools for security testing React Native applications as intended, and therefore these OWASP-provided methodologies are the best there are currently available.

Dynamical analysis and manual testing were proven to be more accurate than statical analysis. There are currently, at the time of writing, no React Native-specific testing tools available. Therefore, when testing React Native, dynamic and manual analyses should be used at least as a part of the testing process. This will allow React Native specifics to be better considered. Therefore, the methodology provided by the master's thesis of Wällstedts [2019] would be sufficient to test React Native applications, as OWASP MASVS [2021] is based mainly on manual steps.

10.2 Limitations

This study was made as a master's thesis. Therefore, there was no budget for commercial security testing tools. Therefore, this study is based on open-source security testing tools and methodologies. However, it is important to acknowledge that multiple commercial tools and automated testing services are available for testing different configurations of mobile and JavaScript applications.

The second limitation of this study was related to the devices available for the study. This is based on the same budget-related limitations as the issue stated earlier. There was no rooted physical iOS device available for dynamically or manually testing iOS-related security issues. This was noticed, when the iOS environment was analyzed by using Frida, and the iOS simulator had to be used. The problem with the usage of the iOS simulator is that it is not an emulator but a simulator, which means that the simulated iOS environment does not fully match the actual iOS device [OWASP, 2020].

10.3 Future work

During this study, it was found that there is currently a general lack of scientific literature regarding security testing modern hybrid mobile applications built with frameworks like React Native. This issue exists, at least in academia. There are, however, some studies conducted regarding security testing hybrid HTML5 based mobile applications by using frameworks like Apache Cordova. These studies are introduced in more detail in the related work section. However, the problem with these studies is that Cordova differs greatly from React Native as it does not use mobile native components but the actual web components [Apache, 2021]. However, this lack of React Native-related security testing studies does not necessarily mean that there is no React Native-related security knowledge in the industry nor academia.

An interesting approach for further studying these issues regarding security testing React Native applications would be to request a statement about suggested methodologies and tools from different stakeholders working around the React Native security. Interesting stakeholders, in my opinion, would be Facebook, the owner of React Native, and different security testing firms conducting security validations of React Native applications. By researching and composing methodologies provided by these stakeholders, a better understanding of the current state of the React Natives security testing and the tools used would be achieved.

The final area of possible work for the future was introduced briefly during the tool section. This is a Frida-based React Native-specific testing toolbox. Frida is a very flexible open-source dynamical analysis tool that supports out-of-the-box all technologies used by React Native [Frida, 2021]. In addition to supporting out-of-the-box all React Natives tools, Frida has a wide variety of functionalities for testing, modifying, and tracing mobile applications' runtime processes [Frida, 2021]. A similar tool was

developed to Cordova framework in the study conducted by Brucker and others [2016]. Like the study stated, Cordova has according to article, issues regarding cross-language calls, in React Native's case this would mean the bridge component. Currently, there is no ready-made toolset for testing React Natives bridge, nor any problems disclosed regarding that. Overall, Frida would make it possible to create this toolbox without implementing low-level processes from the beginning [Frida, 2021].

References

- [Acharay, *et al*, 2015] S. Acharya, B. Ehrenreich and J. Marciniak, "OWASP inspired mobile security," 2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 2015, pp. 782-784, doi: 10.1109/BIBM.2015.7359786.
- [Alanda et al 2020] Alanda A, Satria D, Mooduto H., Kurniawan B. Mobile Application Security Penetration Testing Based on OWASP. In: IOP conference series Materials Science and Engineering. IOP Publishing; 2020. p. 12036–.
- [Apache, 2021] Apache Cordova documentation Accessible electronically, 2021 <https://cordova.apache.org/docs/en/latest/>
- [Apple 2021] Apple developer documentation Accessible electronically <https://developer.apple.com/documentation/> referenced 12.10.2021
- [Arnativich, *et al*, 2018] Y. L. Arnatovich, L. Wang, N. M. Ngo and C. Soh, "A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation," in IEEE Access, vol. 6, pp. 12382-12394, 2018, doi: 10.1109/ACCESS.2018.2808340.
- [Bhandari, *et al*, 2017] Bhandari A, Bhuiyan M, Prasad PW. Enhancement of MD5 Algorithm for Secured Web Development. J. Softw.. 2017 Apr 1;12(4):240-52.
- [Boduch, et al. 2020] Boduch, Adam, and Roy Derks. 2020. React and React Native: A Complete Hands-On Guide to Modern Web and Mobile Development with React. Js, 3rd Edition. Birmingham: Packt Publishing, Limited.
- [Bojjagani, *et al*, 2017] *Bojjagani* S, Sastry VN. VAPT*Ai*: A Threat Model for Vulnerability Assessment and Penetration Testing of Android and iOS Mobile Banking Apps. In: 2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC). IEEE; 2017. p. 77–86.
- [Borja, *et al* 2021] Borja, Thomás, Marco E Benalcázar, Ángel Leonardo Valdivieso Caraguay, and Lorena Isabel Barona López. "Risk Analysis and Android Application Penetration Testing Based on OWASP 2016." In *Information Technology and Systems*, 461–478. Cham: Springer International Publishing, 2021.
- [Burp, 2021] Burp Suite Documentation Accessible electronically <https://portswigger.net/burp/documentation/desktop/tools> referenced 21.09.2021
- [Brucker, *et al*, 2016] Brucker AD, Herzberg M. On the Static Analysis of Hybrid Mobile Apps: A Report on the State of Apache Cordova Nation. In: Engineering Secure Software and Systems. (2016) Springer International Publishing; p. 72–88.
- [Cifuentes, *et al*, 2015] Cifuentes, Y., L. Beltrán, and L. Ramírez. "Analysis of security vulnerabilities for mobile health applications." International Journal of Health and Medical Engineering 9.9 (2015): 1067-1072.
- [CossacLabs, 2021] CossacLabs React Native Security Accessible electronically <https://www.cossacklabs.com/blog/react-native-app-security.html> Referenced 18.10.2021

[Decan, et al 2018] Decan, Alexandre, Tom Mens, and Eleni Constantinou. “On the Impact of Security Vulnerabilities in the Npm Package Dependency Network.” Proceedings of the 15th International Conference on Mining Software Repositories. ACM, 2018. 181–191. Web.

[Dewhurst 2020] Ryan Dewhurst, *OWASP static code analysis* Accessible electronically https://owasp.org/www-community/controls/Static_Code_Analysis referenced 14.09.2021

[Eslint, 2021] Eslint documentation Accessible electronically

<https://eslint.org/docs/developer-guide/architecture> Referenced 19.11.2021

[Expo, 2021] Expo documentation Accessible electronically <https://docs.expo.dev/> referenced 07.09.2021

[Facebook A, 2021] React-native Github page 2021. Accessible electronically

<https://github.com/facebook/react-native> Referenced 07.09.2021.

[Facebook A, 2020] Who’s using React Native? 2020. Accessible electronically

<https://facebook.github.io/react-native/showcase/> Checked 05.09.2021.

[Facebook, 2019] React-Native Security Documentation. Accessible electronically

<https://reactnative.dev/docs/security/> Referenced 05.10.2021.

[Facebook B 2020] Thinking in React Accessible electronically

<https://reactjs.org/docs/thinking-in-react.html> referenced 07.09.2021.

[Facebook B, 2021]] React.Js Documentation 2021. Accessible electronically

<https://reactjs.org/docs/> Referenced 05.09.2021.

[Facebook C, 2021] React Native Architecture Accessible electronically

<https://tkssharma.gitbook.io/react-training/react-native/react-native-architecture> referenced 07.09.2021

[Facebook D, 2021] React Native Documentation Accessible electronically

<https://reactnative.dev/docs/getting-started> referenced 07.09.2021

[Facebook E, 2021] Facebook Github Project Listing Accessible electronically

<https://github.com/facebook> referenced 07.09.2021

[Facebook F, 2021] React Native Internals Accessible electronically

<https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html> referenced 07.09.2021.

[Frida, 2021] Frida documentation Accessible Electronically <https://frida.re/docs>

referenced 23.11.2021

[Frida A, 2021] Frida codeshare Accessible Electronically <https://codeshare.frida.re/>

referenced 11.12.2021

[GitHub pull request statistic 2020] React-Native GitHub 2020. Accessible electronically

<https://madnight.github.io/github/#/pushes/2020/4> Referenced 05.09.2021.

[Github, 2021] About GitHub security advisors Accessible electronically

<https://docs.github.com/en/code-security/security-advisories/about-github-security-advisories> referenced 21.09.2021

[Google, 2021] Android documentation Accessible Electronically

<https://developer.android.com/> referenced 27.11.2021

[Hale and Hanson, 2015] M. L. Hale and S. Hanson, "A Testbed and Process for Analyzing Attack Vectors and Vulnerabilities in Hybrid Mobile Apps Connected to Restful Web Services," 2015 IEEE World Congress on Services, 2015, pp. 181-188, doi: 10.1109/SERVICES.2015.35.

[Hatamian, *et al*, 2021] Hatamian, M., Wairimu, S., Momen, N., & Fritsch, L. (2021). A privacy and security analysis of early-deployed COVID-19 contact tracing Android apps. *Empirical Software Engineering*, 26(3), 1-51.

[Haq and Khan, 2021] I. U. Haq and T. A. Khan, "Penetration Frameworks and Development Issues in Secure Mobile Application Development: A Systematic Literature Review," in *IEEE Access*, vol. 9, pp. 87806-87825, 2021, doi: 10.1109/ACCESS.2021.3088229.

[Helfrich 2019] Helfrich, James 2019 Security for Software Engineers 2019, Security for Software Engineers

[Joseph, *et al*, 2021] Joseph, Ryan B.; ZIBRAN, Minhaz F.; EISHITA, Farjana Z. Choosing the weapon: A comparative study of security analyzers for android applications. In: 2021 IEEE/ACIS 19th International Conference on Software Engineering Research, Management and Applications (SERA). IEEE, 2021. p. 51-57.

[Js-Beautify, 2021] Js-Beautify documentation Accessible electronically

<https://github.com/beautify-web/js-beautify> referenced 11.12.2021

[JScrambler. 2021] JScrambler homepage Accessible electronically

<https://blog.jscrambler.com/how-to-protect-react-native-apps-with-jscrambler> referenced 19.10.2021

[Kohli and Mahsa, 2020] Kohli, Narmada, and Mahsa Mohaghegh. "Security Testing Of Android Based Covid Tracer Applications." 2020 *IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*. IEEE, 2020.

[Lenarduzzi, *et al*, 2019] Lenarduzzi V, Lomio F, Huttunen H, Taibi D. Are SonarQube Rules Inducing Bugs? 2019 Accessible online: <https://arxiv.org/abs/1907.00376> Referenced 7.10.2021

[Linkedin, 2018] Qark Github Page Accessible Electronically

<https://github.com/linkedin/qark> referenced 23.11.2021

[Long, 2016] OWASP Dependency Check presentation Accessible Electronically

<https://jeremylong.github.io/DependencyCheck/general/dependency-check.pdf>

referenced 23.11.2021

[Microsoft, 2021] CodePush Github page Accessible Electronically

<https://github.com/microsoft/react-native-code-push> referenced 2.11.2021

- [MITRE, 2021] Mitre vulnerabilities about Accessible electronically
<https://cve.mitre.org/referenced> 18.10.2021
- [MobSf, 2021] MobSf documentation Accessible electronically
<https://mobsf.github.io/docs/#/> referenced 5.10.2021
- [Mozilla, 2021] Mozilla JavaScript about page Accessible electronically
<https://developer.mozilla.org/en-US/docs/Web/JavaScript> referenced 12.10.2021
- [Mueller, *et al* 2020] Bernhard Mueller, Sven Scheleir, Jeroen Willemsen, Carlos Holguera. OWASP Mobile Security Testing Guide 2020. Accessible electronically
<https://owasp.org/www-project-mobile-security-testing-guide/> Referenced 07.09.2021.
- [NIST, 2021] NIST CVSS documentation Accessible electronically
<https://nvd.nist.gov/vuln-metrics/cvss> referenced 18.10.2021
- [Ogata, *et al*, 2019] Ogata, M. , Franklin, J. , Voas, J. , Sritapan, V. and Quirolgico, S. (2019), Vetting the Security of Mobile Applications, Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, [online], <https://doi.org/10.6028/NIST.SP.800-163r1> (Accessed October 15, 2021)
- [OWASP, 2016] OWASP Mobile Top Ten Documentation 2016. Accessible electronically <https://owasp.org/www-project-mobile-top-10/> Checked 05.09.2021.
- [OWASP, 2019] OWASP methodology Accessible electronically https://owasp.org/www-project-top-ten/2017/Methodology_and_Data.html Referenced 07.09.2021.
- [OWASP, 2020] OWASP, About the OWASP Foundation Accessible electronically
<https://owasp.org/about> referenced 07.09.2021.
- [Palacios, *et al*, 2019] Palacios, José, Gabriel López, and Franklin Sánchez. “Security Analysis Protocol for Android-Based Mobile Applications.” RISTI : Revista Ibérica de Sistemas e Tecnologias de Informação 2019, no. 19 (2019): 366–378
- [PTES, 2021] Ptes Documentation Accessible electronically http://www.pentest-standard.org/index.php/Main_Page referenced 15.10.2021
- [Redux, 2021] Redux documentation Accessible Electronically
<https://redux.js.org/introduction/getting-started> Referenced 2.11.2021
- [Qark, 2019] QARK issue Accessible electronically
<https://github.com/linkedin/qark/issues/355> referenced 11.12.2021
- [Rguez-Sánchez, 2019] Secure your React Native Apps before production! Accessible electronically <https://manuelrdsg.github.io/2019/06/secure-your-react-native-apps-before-production/> referenced 19.10.2021
- [Rodríguez, *et al*, 2018] Rodríguez, Jesús & Gil, Celio & Baquero Rey, Luis & Hernandez Bejarano, Miguel. (2018). A Conceptual Exploration for the Safe Development of Mobile Devices Software Based on OWASP. International Journal of Applied Engineering Research. 13. 13603-13609.

[Shinde and Ardhapurkar 2016] P. S. Shinde and S. B. Ardhapurkar, "Cyber security analysis using vulnerability assessment and penetration testing," *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, 2016, pp. 1-5, doi: 10.1109/STARTUP.2016.7583912.

[SonarQube, 2021] SonarQube documentation Accessible electronically <https://docs.sonarqube.org/latest/> referenced 30.09.2021

[Sugandh 2015] Shah, Sugandh, and B M Mehtre. "An Overview of Vulnerability Assessment and Penetration Testing Techniques." *Journal of computer virology and hacking techniques* 11, no. 1 (2015): 27–49.

[Stackoverflow A, 2019] React Native obfuscation Accessible electronically <https://stackoverflow.com/questions/52696998/obfuscate-entire-react-native-app-including-javascript-code> referenced 19.10.2021

[Tuni, 2021] Andor databases, Accessible online: <https://libguides.tuni.fi/az.php?referenced> 8.10.2021

[Umro, et al 2012] UMRAO, SACHIN; KAUR, MANDEEP; GUPTA, GOVIND KUMAR. Vulnerability assessment and penetration testing. *International Journal of Computer & Communication Technology*, 2012, 3.6-8: 71-74.

[Viljay Kumar 2016] Velu, Vijay Kumar (2016) *Mobile application penetration testing: explore real-world threat scenarios attacks on mobile applications, and ways to counter them*. 1st ed. PACKT Publishing.

[Vondráček, et al 2018] Vondráček, Martin, Jan Pluskal, and Ondřej Ryšavý. "Automated Man-in-the-Middle Attack Against Wi-Fi Networks." *The journal of digital forensics, security and law* 13.1 (2018): 59–80. Web.

[Wang and Alshboul, 2015] Yong Wang & Alshboul, Y. (2015) 'Mobile security testing approaches and challenges', in 2015 First Conference on Mobile and Secure Services (MOBISECSERV). [Online]. 2015 IEEE. pp. 1–5.

[Weidman, 20021] Adam Weidman, Owasp ReDos documentation Accessible electronically <https://security.snyk.io/vuln/SNYK-JS-ANSIHTML-1296849> Referenced 19.11.2021

[Willemsen, et al, 2021] J Willemsen, B Mueller, R Atefinia, J Beckers, et al OWASP Mobile Security Verification Standard Accessible electronically <https://mobile-security.gitbook.io/masvs/> Referenced 4.11.2021

[Wällstedt, 2019] Wällstedt V. Implementation and Security Evaluation of User-Customized Content in a Mobile Application [Internet] [Dissertation]. 2019. Available from: <http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-158495>

[Zhang, et al, 2021] Zhang, X., Breitinger, F., Luechinger, E., O'Shaughnessy, S. (2021). Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39, 301285.

[Zimmermann, et al, 2019] Zimmermann, Markus et al. "Small World with High Risks: A Study of Security Threats in the Npm Ecosystem." (2019): n. pag