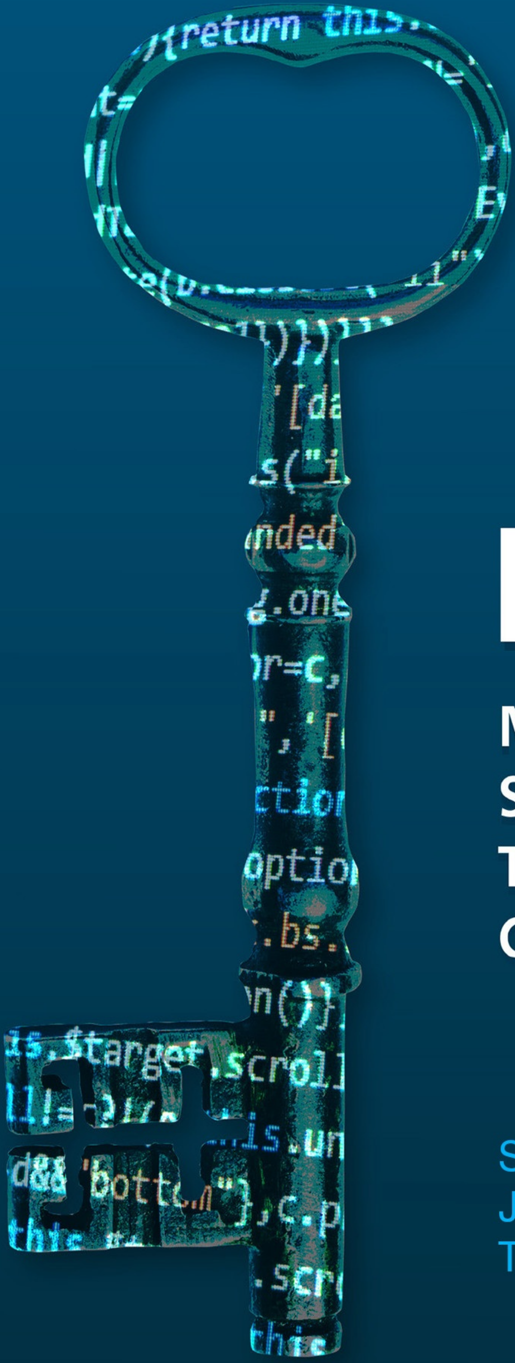
OWASP
Open Web Application Security Project



MSTG

MOBILE
SECURITY
TESTING
GUIDE

Sven Schleier
Jeroen Willemsen
The OWASP mobile team

VERSION 1.0.1

Table of Contents

Foreword

Welcome to the OWASP Mobile Security Testing Guide. Feel free to explore the existing content, but do note that it may change at any time. New APIs and best practices are introduced in iOS and Android with every major (and minor) release and also vulnerabilities are found every day.

If you have feedback or suggestions, or want to contribute, create an issue on GitHub or ping us on Slack. See the README for instructions:

https://www.github.com/OWASP/owasp-mstg/

---

squirrel (noun plural): Any arboreal sciurine rodent of the genus Sciurus, such as S. vulgaris (red squirrel) or S. carolinensis (grey squirrel), having a bushy tail and feeding on nuts, seeds, etc.

On a beautiful summer day, a group of ~7 young men, a woman, and approximately three squirrels met in a Woburn Forest villa during the OWASP Security Summit 2017. So far, nothing unusual. But little did you know, within the next five days, they would redefine not only mobile application security, but the very fundamentals of book writing itself (ironically, the event took place near Bletchley Park, once the residence and work place of the great Alan Turing).

Or maybe that's going to far. But at least, they produced a proof-of-concept for an unusual security book. The Mobile Security Testing Guide (MSTG) is an open, agile, crowd-sourced effort, made of the contributions of dozens of authors and reviewers from all over the world.

Because this isn't a normal security book, the introduction doesn't list impressive facts and data proving importance of mobile devices in this day and age. It also doesn't explain how mobile application security is broken, and why a book like this was sorely needed, and the authors don't thank their wifes and friends without whom the book wouldn't have been possible.

We do have a message to our readers however! The first rule of the OWASP Mobile Security Testing Guide is: Don't just follow the OWASP Mobile Security Testing Guide. True excellence at mobile application security requires a deep understanding of mobile operating systems, coding, network security, cryptography, and a whole lot of other things, many of which we can only touch on briefly in this book. Don't stop at security testing. Write your own apps, compile your own kernels, dissect mobile malware, learn how things tick. And as you keep learning new things, consider contributing to the MSTG yourself! Or, as they say: "Do a pull request".

OWASP
SUMMIT
2017

Frontispiece

About the OWASP Mobile Security Testing Guide

The OWASP Mobile Security Testing Guide (MSTG) is a comprehensive manual for testing the security of mobile apps. It describes processes and techniques for verifying the requirements listed in the Mobile Application Security Verification Standard (MASVS), and provides a baseline for complete and consistent security tests.

OWASP thanks the many authors, reviewers, and editors for their hard work in developing this guide. If you have any comments or suggestions on the Mobile Security Testing Guide, please join the discussion around MASVS and MSTG in the OWASP Mobile Security Project Slack Channel. You can sign up for the Slack channel yourself using this invite. (Please open a PR if the invite has expired.)

Copyright and License

Acknowledgements

Note: This contributor table is generated based on our GitHub contribution statistics. For more information on these stats, see the GitHub Repository README. We manually update the table, so be patient if you're not listed immediately.

Authors

Bernhard Mueller

Bernhard is a cyber security specialist with a talent for hacking systems of all kinds. During more than a decade in the industry, he has published many zero-day exploits for software such as MS SQL Server, Adobe Flash Player, IBM Director, Cisco VOIP, and ModSecurity. If you can name it, he has probably broken it at least once. BlackHat USA commended his pioneering work in mobile security with a Pwnie Award for Best Research.

Sven Schleier

Sven is an experienced web and mobile penetration tester and assessed everything from historic Flash applications to progressive mobile apps. He is also a security engineer that supported many projects end-to-end during the SDLC to "build security in". He was speaking at local and international meetups and conferences and is conducting hands-on workshops about web application and mobile app security.

Jeroen Willemsen

Jeroen is a principal security architect at Xebia with a passion for mobile security and risk management. He has supported companies as a security coach, a security engineer and as a full-stack developer, which makes him a jack of all trades. He loves explaining technical subjects: from security issues to programming challenges.

Co-Authors

Co-authors have consistently contributed quality content and have at least 2,000 additions logged in the GitHub repository.

Romuald Szkudlarek

Romuald is a passionate cyber security & privacy professional with over 15 years of experience in the web, mobile, IoT and cloud domains. During his career, he has been dedicating his spare time to a variety of projects with the goal of advancing the sectors of software and security. He is teaching regularly at various institutions. He holds CISSP, CCSP, CSSLP, and CEH credentials.

Top Contributors

Top contributors have consistently contributed quality content and have at least 500 additions logged in the GitHub repository.

- Pawel Rzepa
- Francesco Stillavato
- Henry Hoggard
- Andreas Happe
- Kyle Benac
- Alexander Anthuk
- Wen Bin Kong
- Abdessamad Temmar
- Bolot Kerimbaev
- Cláudio André
- Slawomir Kosowski

Contributors

Contributors have contributed quality content and have at least 50 additions logged in the GitHub repository.

Jin Kung Ong, Sjoerd Langkemper, Gerhard Wagner, Michael Helwig, Pece Milosev, Ryan Teoh, Denis Pilipchuk, Jeroen Beckers, Dharshin De Silva, Anatoly Rosencrantz, Abhinav Sejpal, Daniel Ramirez Martin, Enrico Verzegnassi, Yogesh Sharma, Dominique Righetto, Raul Siles, Nick Epson, Prathan Phongthiproek, Tom Welch, Luander Ribeiro, Dario Incalza, Akanksha Bana, Oguzhan Topgul, Carlos Holguera, Vikas Gupta, David Fern, Pishu Mahtani, Anuruddha E.

Reviewers

Reviewers have consistently provided useful feedback through GitHub issues and pull request comments.

- Sjoerd Langkemper
- Anant Shrivastava

Editors

- Heaven Hodges

- Caitlin Andrews
- Nick Epson
- Anita Diamond
- Anna Szkudlarek

Others

Many other contributors have committed small amounts of content, such as a single word or sentence (less than 50 additions). The full list of contributors is available on GitHub.

Sponsors

While both the MASVS and the MSTG are created and maintained by the community on a voluntary basis, sometimes a little bit of outside help is required. We therefore thank our sponsors for providing the funds to be able to hire technical editors. Note that their sponsorship does not influence the content of the MASVS or MSTG in any way. The sponsorship packages are described on the OWASP Project Wiki.

Honourable Benefactor

NowSecure™

Older Versions

The Mobile Security Testing Guide was initiated by Milan Singh Thakur in 2015. The original document was hosted on Google Drive. Guide development was moved to GitHub in October 2016.

OWASP MSTG "Beta 2" (Google Doc)

Authors	Reviewers	Top Contributors
Milan Singh Thakur, Abhinav Sejpal, Blessen Thomas, Dennis Titze, Davide Cioccia, Pragati Singh, Mohammad Hamed Dadpour, David Fern, Mirza Ali, Rahil Parikh, Anant Shrivastava, Stephen Corbiaux, Ryan Dewhurst, Anto Joseph, Bao Lee, Shiv Patel, Nutan Kumar Panda, Julian Schütte, Stephanie Vanroelen, Bernard Wagner, Gerhard Wagner, Javier Dominguez	Andrew Muller, Jonathan Carter, Stephanie Vanroelen, Milan Singh Thakur	Jim Manico, Paco Hope, Pragati Singh, Yair Amit, Amin Lalji, OWASP Mobile Team

OWASP MSTG "Beta 1" (Google Doc)

Authors	Reviewers	Top Contributors
Milan Singh Thakur, Abhinav Sejpal, Pragati Singh, Mohammad Hamed Dadpour, David Fern, Mirza Ali, Rahil Parikh	Andrew Muller, Jonathan Carter	Jim Manico, Paco Hope, Yair Amit, Amin Lalji, OWASP Mobile Team

Overview

Introduction to the OWASP Mobile Security Testing Guide

New technology always introduces new security risks, and mobile computing is no exception. Security concerns for mobile apps differ from traditional desktop software in some important ways. Modern mobile operating systems are arguably more secure than traditional desktop operating systems, but problems can still appear when we don't carefully consider security during mobile app development. Data storage, inter-app communication, proper usage of cryptographic APIs, and secure network communication are only some of these considerations.

Key Areas in Mobile Application Security

Many mobile app penetration testing tools have a background in network and web app penetration testing, a quality that is valuable for mobile app testing. Almost every mobile app talks to a back-end service, and those services are prone to the same types of attacks we are familiar with in web apps on desktop machines. Mobile apps differ in that there is a smaller attack surface and therefore more security against injection and similar attacks. Instead, we must prioritize data protection on the device and the network to increase mobile security.

Let's discuss the key areas in mobile app security.

Local Data Storage

The protection of sensitive data, such as user credentials and private information, is crucial to mobile security. If an app uses operating system APIs such as local storage or inter-process communication (IPC) improperly, the app might expose sensitive data to other apps running on the same device. It may also unintentionally leak data to cloud storage, backups, or the keyboard cache. Additionally, mobile devices can be lost or stolen more easily compared to other types of devices, so it's more likely an individual can gain physical access to the device, making it easier to retrieve the data.

When developing mobile apps, we must take extra care when storing user data. For example, we can use appropriate key storage APIs and take advantage of hardware-backed security features when available.

Fragmentation is a problem we deal with especially on Android devices. Not every Android device offers hardware-backed secure storage, and many devices are running outdated versions of Android. For an app to be supported on these out-of-date devices, it would have to be created using an older version of Android's API which may lack important security features. For maximum security, the best choice is to create apps with the current API version even though that excludes some users.

Communication with Trusted Endpoints

Mobile devices regularly connect to a variety of networks, including public WiFi networks shared with other (potentially malicious) clients. This creates opportunities for a wide variety of network-based attacks ranging from simple to complicated and old to new. It's crucial to maintain the confidentiality and integrity of information exchanged between the mobile app and remote service endpoints. As a basic requirement, mobile apps must set up a secure, encrypted channel for network communication using the TLS protocol with appropriate settings.

Authentication and Authorization

In most cases, sending users to log in to a remote service is an integral part of the overall mobile app architecture. Even though most of the authentication and authorization logic happens at the endpoint, there are also some implementation challenges on the mobile app side. Unlike web apps, mobile apps often store long-time session tokens that are unlocked with user-to-device authentication features such as fingerprint scanning. While this allows for a quicker login and better user experience (nobody likes to enter complex passwords), it also introduces additional complexity and room for error.

Mobile app architectures also increasingly incorporate authorization frameworks (such as OAuth2) that delegate authentication to a separate service or outsource the authentication process to an authentication provider. Using OAuth2 allows the client-side authentication logic to be outsourced to other apps on the same device (e.g. the system browser). Security testers must know the advantages and disadvantages of different possible authorization frameworks and architectures.

Interaction with the Mobile Platform

Mobile operating system architectures differ from classical desktop architectures in important ways. For example, all mobile operating systems implement app permission systems that regulate access to specific APIs. They also offer more (Android) or less rich (iOS) inter-process communication (IPC) facilities that enable apps to exchange signals and data. These platform-specific features come with their own set of pitfalls. For example, if IPC APIs are misused, sensitive data or functionality might be unintentionally exposed to other apps running on the device.

Code Quality and Exploit Mitigation

Traditional injection and memory management issues aren't often seen in mobile apps due to the smaller attack surface. Mobile apps mostly interact with the trusted backend service and the UI, so even if many buffer overflow vulnerabilities exist in the app, those vulnerabilities usually don't open up any useful attack vectors. The same applies to browser exploits such as cross-site scripting (XSS allows attackers to inject scripts into web pages) that are very prevalent in web apps. However, there are always exceptions. XSS is theoretically possible on mobile in some cases, but it's very rare to see XSS issues that an individual can exploit. For more information about XSS, see Testing for Cross-Site Scripting Flaws in the chapter Testing Code Quality.

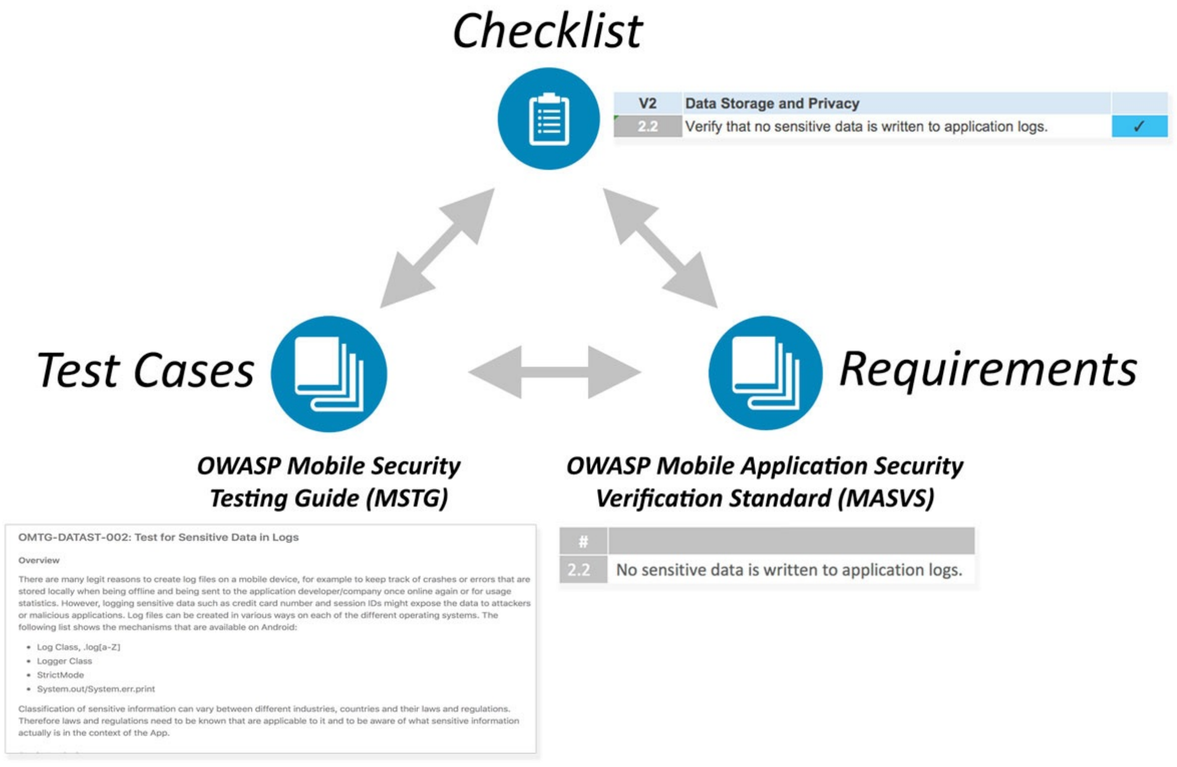
This protection from injection and memory management issues doesn't mean that app developers can get away with writing sloppy code. Following security best practices results in hardened (secure) release builds that are resilient against tampering. Free security features offered by compilers and mobile SDKs help increase security and mitigate attacks.

Anti-Tampering and Anti-Reversing

There are three things you should never bring up in polite conversations: religion, politics, and code obfuscation. Many security experts dismiss client-side protections outright. However, software protection controls are widely used in the mobile app world, so security testers need ways to deal with these protections. We believe there's a benefit to client-side protections if they are employed with a clear purpose and realistic expectations in mind and aren't used to replace security controls.

The OWASP Mobile AppSec Verification Standard

This guide is closely related to the OWASP Mobile Application Security Verification Standard (MASVS). The MASVS defines a mobile app security model and lists generic security requirements for mobile apps. It can be used by architects, developers, testers, security professionals, and consumers to define and understand the qualities of a secure mobile app. The MSTG maps to the same basic set of security requirements offered by the MASVS and depending on the context they can be used individually or combined to achieve different objectives.


Checklist
Test Cases
Requirements
OWASP Mobile Security Testing Guide (MSTG)
OWASP Mobile Application Security Verification Standard (MASVS)

V2	Data Storage and Privacy	
2.2	Verify that no sensitive data is written to application logs.	✓

OMTG-DATAST-002: Test for Sensitive Data in Logs

Overview

There are many legit reasons to create log files on a mobile device, for example to keep track of crashes or errors that are stored locally when being offline and being sent to the application developer/company once online again or for usage statistics. However, logging sensitive data such as credit card number and session IDs might expose the data to attackers or malicious applications. Log files can be created in various ways on each of the different operating systems. The following list shows the mechanisms that are available on Android:

- Log Class, .log[a-Z]
- Logger Class
- StrictMode
- System.out/System.err.print

Classification of sensitive information can vary between different industries, countries and their laws and regulations. Therefore laws and regulations need to be known that are applicable to it and to be aware of what sensitive information actually is in the context of the App.

#	
2.2	No sensitive data is written to application logs.

For example, the MASVS requirements can be used in an app's planning and architecture design stages while the checklist and testing guide may serve as a baseline for manual security testing or as a template for automated security tests during or after development. In the Mobile App Security Testing chapter we'll describe how you can apply the checklist and MSTG to a mobile app penetration test.

Navigating the Mobile Security Testing Guide

The MSTG contains descriptions of all requirements specified in the MASVS. The MSTG contains the following main sections:

1. The General Testing Guide contains a mobile app security testing methodology and general vulnerability analysis techniques as they apply to mobile app security. It also contains additional technical test cases that are OS-independent, such as authentication and session management, network communications, and cryptography.
2. The Android Testing Guide covers mobile security testing for the Android platform, including security basics, security test cases, reverse engineering techniques and prevention, and tampering techniques and prevention.
3. The iOS Testing Guide covers mobile security testing for the iOS platform, including an overview of the iOS OS, security testing, reverse engineering techniques and prevention, and tampering techniques and prevention.

General Testing Guide

Mobile App Taxonomy

The term "mobile app" refers to a self-contained computer program designed to execute on a mobile device. Today, the Android and iOS operating systems cumulatively comprise more than 99% of the mobile OS market share. Additionally, mobile internet usage has surpassed desktop usage for the first time in history, making mobile browsing and apps the most widespread kind of internet-capable applications.

> In this guide, we'll use the term "app" as a general term for referring to any kind of application running on popular mobile OSes.

In a basic sense, apps are designed to run either directly on the platform for which they're designed, on top of a smart device's mobile browser, or using a mix of the two. Throughout the following chapter, we will define characteristics that qualify an app for its respective place in mobile app taxonomy as well as discuss differences for each variation.

Native App

Mobile operating systems, including Android and iOS, come with a Software Development Kit (SDK) for developing applications specific to the OS. Such applications are referred to as *native* to the system for which they have been developed. When discussing an app, the general assumption is that it is a native app implemented in a standard programming language for the respective operating system - Objective-C or Swift for iOS, and Java or Kotlin for Android.

Native apps inherently have the capability to provide the fastest performance with the highest degree of reliability. They usually adhere to platform-specific design principles (e.g. the Android Design Principles), which tends to result in a more consistent user interface (UI) compared to *hybrid* or *web* apps. Due to their close integration with the operating system, native apps can directly access almost every component of the device (camera, sensors, hardware-backed key stores, etc.).

Some ambiguity exists when discussing *native apps* for Android as the platform provides two development kits - the Android SDK and the Android NDK. The SDK, which is based on the Java and Kotlin programming language, is the default for developing apps. The NDK (or Native Development Kit) is a C/C++ development kit used for developing binary libraries that can directly access lower level APIs (such as OpenGL). These libraries can be included in regular apps built with the SDK. Therefore, we say that Android *native apps* (i.e. built with the SDK) may have *native* code built with the NDK.

The most obvious downside of *native apps* is that they target only one specific platform. To build the same app for both Android and iOS, one needs to maintain two independent code bases, or introduce often complex development tools to port a single code base to two platforms (e.g. Xamarin).

Web App

Mobile web apps (or simply, *web apps*) are websites designed to look and feel like a *native app*. These apps run on top of a device's browser and are usually developed in HTML5, much like a modern webpage. Launcher icons may be created to parallel the same feel of accessing a *native app*; however, these icons are essentially the same as a browser bookmark, simply opening the default web browser to load the referenced web page.

Web apps have limited integration with the general components of the device as they run within the confines of a browser (i.e. they are "sandboxed") and usually lack in performance compared to native apps. Since a web app typically targets multiple platforms, their UIs do not follow some of the design principles of a specific platform. The biggest advantage is reduced development and maintenance costs associated with a single code base as well as enabling developers to distribute updates without engaging the platform-specific app stores. For example, a change to the HTML file for a web app can serve as viable, cross-platform update whereas an update to a store-based app requires considerably more effort.

Hybrid App

Hybrid apps attempt to fill the gap between *native* and *web apps*. A *hybrid app* executes like a *native app*, but a majority of the processes rely on web technologies, meaning a portion of the app runs in an embedded web browser (commonly called “webview”). As such, hybrid apps inherit both pros and cons of *native* and *web apps*.

A web-to-native abstraction layer enables access to device capabilities for *hybrid apps* not accessible to a pure *web app*. Depending on the framework used for development, one code base can result in multiple applications that target different platforms, with a UI closely resembling that of the original platform for which the app was developed.

Following is a non-exhaustive list of more popular frameworks for developing *hybrid apps*:

- Apache Cordova
- Framework 7
- Ionic
- jQuery Mobile
- Native Script
- Onsen UI
- React Native
- Sencha Touch

Progressive Web App

Progressive Web Apps (PWA) load like regular web pages, but differ from usual web apps in several ways. For example it's possible to work offline and access to mobile device hardware is possible, that traditionally is only available to native mobile apps.

PWAs combine different open standards of the web offered by modern browsers to provide benefits of a rich mobile experience. A Web App Manifest, which is a simple JSON file, can be used to configure the behaviour of the app after "installation".

PWAs are supported by Android and iOS, but not all hardware features are yet available. For example Push Notifications, Face ID on iPhone X or ARKit for augmented reality is not available yet on iOS. An overview of PWA and supported features on each platform can be found in a Medium article from Maximiliano Firtman.

What's Covered in the Mobile Testing Guide?

Throughout this guide, we will focus on apps for the two platforms dominating the market: Android and iOS. Mobile devices are currently the most common device class running these platforms – increasingly however, the same platforms (in particular, Android) run on other devices, such as smartwatches, TVs, car navigation/audio systems, and other embedded systems.

Given the vast amount of mobile app frameworks available it would be impossible to cover all of them exhaustively. Therefore, we focus on *native* apps on each operating system. However, the same techniques are also useful when dealing with web or hybrid apps (ultimately, no matter the framework, every app is based on native components).

Mobile App Security Testing

In the following sections we'll provide a brief overview of general security testing principles and key terminology. The concepts introduced are largely identical to those found in other types of penetration testing, so if you are an experienced tester you may be familiar with some of the content.

Throughout the guide, we use "mobile app security testing" as a catchall phrase to refer to the evaluation of mobile app security via static and dynamic analysis. Terms such as "mobile app penetration testing" and "mobile app security review" are used somewhat inconsistently in the security industry, but these terms refer to roughly the same thing. A mobile app security test is usually part of a larger security assessment or penetration test that encompasses the client-server architecture and server-side APIs used by the mobile app.

In this guide, we cover mobile app security testing in two contexts. The first is the "classical" security test completed near the end of the development life cycle. In this context, the tester accesses a nearly finished or production-ready version of the app, identifies security issues, and writes a (usually devastating) report. The other context is characterized by the implementation of requirements and the automation of security tests from the beginning of the software development life cycle onwards. The same basic requirements and test cases apply to both contexts, but the high-level method and the level of client interaction differ.

Principles of Testing

White-box Testing versus Black-box Testing

Let's start by defining the concepts:

- Black-box testing is conducted without the tester's having any information about the app being tested. This process is sometimes called "zero-knowledge testing." The main purpose of this test is allowing the tester to behave like a real attacker in the sense of exploring possible uses for publicly available and discoverable information.
- White-box testing (sometimes called "full knowledge testing") is the total opposite of black-box testing in the sense that the tester has full knowledge of the app. The knowledge may encompass source code, documentation, and diagrams. This approach allows much faster testing than black-box testing due to it's transparency and with the additional knowledge gained a tester can build much more sophisticated and granular test cases.
- Gray-box testing is all testing that falls in between the two aforementioned testing types: some information is provided to the tester (usually credentials only), and other information is intended to be discovered. This type of testing is an interesting compromise in the number of test cases, the cost, the speed, and the scope of testing. Gray-box testing is the most common kind of testing in the security industry.

We strongly advise that you request the source code so that you can use the testing time as efficiently as possible. The tester's code access obviously doesn't simulate an external attack, but it simplifies the identification of vulnerabilities by allowing the tester to verify every identified anomaly or suspicious behavior at the code level. A white-box test is the way to go if the app hasn't been tested before.

Even though decompiling on Android is straightforward, the source code may be obfuscated, and de-obfuscating will be time-consuming. Time constraints are therefore another reason for the tester to have access to the source code.

Vulnerability Analysis

Vulnerability analysis is usually the process of looking for vulnerabilities in an app. Although this may be done manually, automated scanners are usually used to identify the main vulnerabilities. Static and dynamic analysis are types of vulnerability analysis.

Static versus Dynamic Analysis

Static Application Security Testing (SAST) involves examining an application's components without executing them, by analyzing the source code either manually or automatically. OWASP provides information about Static Code Analysis that may help you understand techniques, strengths, weaknesses, and limitations.

Dynamic Application Security Testing (DAST) involves examining the app during runtime. This type of analysis can be manual or automatic. It usually doesn't provide the information that static analysis provides, but it is a good way to detect interesting elements (assets, features, entry points, etc.) from a user's point of view.

Now that we have defined static and dynamic analysis, let's dive deeper.

Static Analysis

During static analysis, the mobile app's source code is reviewed to ensure appropriate implementation of security controls. In most cases, a hybrid automatic/manual approach is used. Automatic scans catch the low-hanging fruit, and the human tester can explore the code base with specific usage contexts in mind.

Manual Code Review

A tester performs manual code review by manually analyzing the mobile application's source code for security vulnerabilities. Methods range from a basic keyword search via the 'grep' command to a line-by-line examination of the source code. IDEs (Integrated Development Environments) often provide basic code review functions and can be extended with various tools.

A common approach to manual code analysis entails identifying key security vulnerability indicators by searching for certain APIs and keywords, such as database-related method calls like "executeStatement" or "executeQuery". Code containing these strings is a good starting point for manual analysis.

In contrast to automatic code analysis, manual code review is very good for identifying vulnerabilities in the business logic, standards violations, and design flaws, especially when the code is technically secure but logically flawed. Such scenarios are unlikely to be detected by any automatic code analysis tool.

A manual code review requires an expert code reviewer who is proficient in both the language and the frameworks used for the mobile application. Full code review can be a slow, tedious, time-consuming process for the reviewer, especially given large code bases with many dependencies.

Automated Source Code Analysis

Automated analysis tools can be used to speed up the review process of Static Application Security Testing (SAST). They check the source code for compliance with a predefined set of rules or industry best practices, then typically display a list of findings or warnings and flags for all detected violations. Some static analysis tools run against the compiled app only, some must be fed the original source code, and some run as live-analysis plugins in the Integrated Development Environment (IDE).

Although some static code analysis tools incorporate a lot of information about the rules and semantics required to analyze mobile apps, they may produce many false positives, particularly if they are not configured for the target environment. A security professional must therefore always review the results.

The chapter "Testing tools" includes a list of static analysis tools, which can be found at the end of this book.

Dynamic Analysis

The focus of DAST is the testing and evaluation of apps via their real-time execution. The main objective of dynamic analysis is finding security vulnerabilities or weak spots in a program while it is running. Dynamic analysis is conducted both at the mobile platform layer and against the back-end services and APIs, where the mobile app's request and response patterns can be analyzed.

Dynamic analysis is usually used to check for security mechanisms that provide sufficient protection against the most prevalent types of attack, such as disclosure of data in transit, authentication and authorization issues, and server configuration errors.

Avoiding False Positives

Automated Scanning Tools

Automated testing tools' lack of sensitivity to app context is a challenge. These tools may identify a potential issue that's irrelevant. Such results are called "false positives".

For example, security testers commonly report vulnerabilities that are exploitable in a web browser but aren't relevant to the mobile app. This false positive occurs because automated tools used to scan the back-end service are based on regular browser-based web applications. Issues such as CSRF (Cross-site Request Forgery) and Cross-Site Scripting (XSS) are reported accordingly.

Let's take CSRF as an example. A successful CSRF attack requires the following:

- The ability to entice the logged-in user to open a malicious link in the web browser used to access the vulnerable site.
- The client (browser) must automatically add the session cookie or other authentication token to the request.

Mobile apps don't fulfil these requirements: even if WebViews and cookie-based session management are used, any malicious link the user clicks opens in the default browser, which has a separate cookie store.

Stored Cross-Site Scripting (XSS) can be an issue if the app includes WebViews, and it may even lead to command execution if the app exports JavaScript interfaces. However, reflected Cross-Site Scripting is rarely an issue for the reason mentioned above (even though whether they should exist at all is arguable — escaping output is simply a best practice).

> In any case, consider exploit scenarios when you perform the risk assessment; don't blindly trust your scanning tool's output.

Clipboard

When typing data into input fields, the clipboard can be used to copy in data. The clipboard is accessible system-wide and is therefore shared by apps. This sharing can be misused by malicious apps to get sensitive data that has been stored in the clipboard.

Before iOS 9, a malicious app might monitor the pasteboard in the background while periodically retrieving `[UIPasteboard generalPasteboard].string`. As of iOS 9, pasteboard content is accessible to apps in the foreground only, which reduces the attack surface of password sniffing from the clipboard dramatically.

For Android there was a PoC exploit released in order to demonstrate the attack vector if passwords are stored within the clipboard. Disabling pasting in passwords input fields was a requirement in the MASVS 1.0, but was removed due to several reasons:

- Preventing pasting into input fields of an app, does not prevent that a user will copy sensitive information anyway. Since the information has already been copied before the user notices that it's not possible to paste it in, a malicious app has already sniffed the clipboard.
- If pasting is disabled on password fields users might even choose weaker passwords that they can remember and they cannot use password managers anymore, which would contradict the original intention of making the app more secure.

When using an app you should still be aware that other apps are reading the clipboard continuously, as the Facebook app did. Still, copy-pasting passwords is a security risk you should be aware of, but also cannot be solved by an app.

Penetration Testing (a.k.a. Pentesting)

The classic approach involves all-around security testing of the app's final or near-final build, e.g., the build that's available at the end of the development process. For testing at the end of the development process, we recommend the Mobile App Security Verification Standard (MASVS) and the associated checklist as baseline for testing. A typical security test is structured as follows:

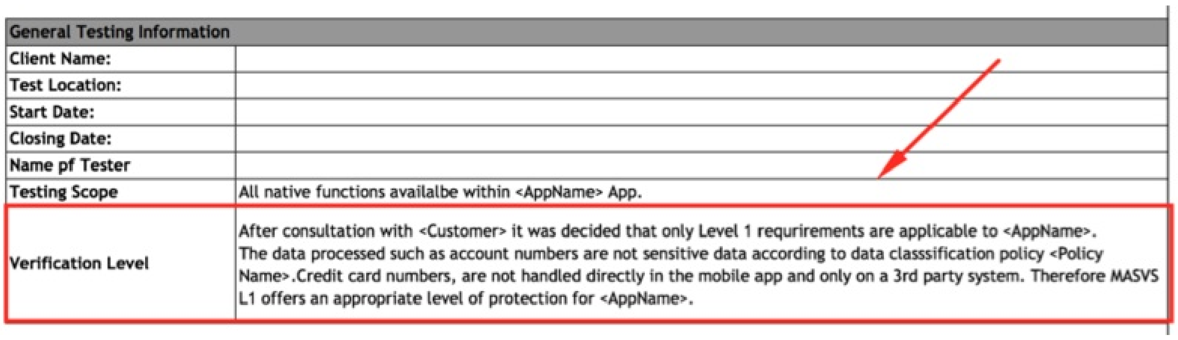
- Preparation - defining the scope of security testing, including identifying applicable security controls, the organization's testing goals, and sensitive data. More generally, preparation includes all synchronization with the client as well as legally protecting the tester (who is often a third party). Remember, attacking a system without written authorization is illegal in many parts of the world!
- Intelligence Gathering - analyzing the environmental and architectural context of the app to gain a general contextual understanding.
- Mapping the Application - based on information from the previous phases; may be complemented by automated scanning and manually exploring the app. Mapping provides a thorough understanding of the app, its entry points, the data it holds, and the main potential vulnerabilities. These vulnerabilities can then be ranked according to the damage their exploitation would cause so that the security tester can prioritize them. This phase includes the creation of test cases that may be used during test execution.
- Exploitation - in this phase, the security tester tries to penetrate the app by exploiting the vulnerabilities identified during the previous phase. This phase is necessary for determining whether vulnerabilities are real and true positives.
- Reporting - in this phase, which is essential to the client, the security tester reports the vulnerabilities he or she has been able to exploit and documents the kind of compromise he or she has been able to perform, including the compromise's scope (for example, the data the tester has been able to access illegitimately).

Preparation

The security level at which the app will be tested must be decided before testing. The security requirements should be decided at the beginning of the project. Different organizations have different security needs and resources available for investing in test activities. Although the controls in MASVS Level 1 (L1) are applicable to all mobile apps, walking through the entire checklist of L1 and Level 2 (L2) MASVS controls with technical and business stakeholders is a good way to decide on a level of test coverage.

Organizations may have different regulatory and legal obligations in certain territories. Even if an app doesn't handle sensitive data, some L2 requirements may be relevant (because of industry regulations or local laws). For example, two-factor authentication (2FA) may be obligatory for a financial app and enforced by a country's central bank and/or financial regulatory authorities.

Security goals/controls defined earlier in the development process may also be reviewed during the discussion with stakeholders. Some controls may conform to MASVS controls, but others may be specific to the organization or application.



General Testing Information	
Client Name:	
Test Location:	
Start Date:	
Closing Date:	
Name pf Tester	
Testing Scope	All native functions availalbe within <AppName> App.
Verification Level	After consultation with <Customer> it was decided that only Level 1 requrirements are applicable to <AppName>. The data processed such as account numbers are not sensitive data according to data classsification policy <Policy Name>.Credit card numbers, are not handled directly in the mobile app and only on a 3rd party system. Therefore MASVS L1 offers an appropriate level of protection for <AppName>.

All involved parties must agree on the decisions and the scope in the checklist because these will define the baseline for all security testing.

Coordinating with the Client

Setting up a working test environment can be a challenging task. For example, restrictions on the enterprise wireless access points and networks may impede dynamic analysis performed at client premises. Company policies may prohibit the use of rooted phones or (hardware and software) network testing tools within enterprise networks. Apps that implement root detection and other reverse engineering countermeasures may significantly increase the work required for further analysis.

Security testing involves many invasive tasks, including monitoring and manipulating the mobile app's network traffic, inspecting the app data files, and instrumenting API calls. Security controls, such as certificate pinning and root detection, may impede these tasks and dramatically slow testing down.

To overcome these obstacles, you may want to request two of the app's build variants from the development team. One variant should be a release build so that you can determine whether the implemented controls are working properly and can be bypassed easily. The second variant should be a debug build for which certain security controls have been deactivated. Testing two different builds is the most efficient way to cover all test cases.

Depending on the scope of the engagement, this approach may not be possible. Requesting both production and debug builds for a white-box test will help you complete all test cases and clearly state the app's security maturity. The client may prefer that black-box tests be focused on the production app and the evaluation of its security controls' effectiveness.

The scope of both types of testing should be discussed during the preparation phase. For example, whether the security controls should be adjusted should be decided before testing. Additional topics are discussed below.

Identifying Sensitive Data

Classifications of sensitive information differ by industry and country. In addition, organizations may take a restrictive view of sensitive data, and they may have a data classification policy that clearly defines sensitive information.

There are three general states from which data may be accessible:

- At rest - the data is sitting in a file or data store
- In use - an application has loaded the data into its address space
- In transit - data has been exchanged between mobile app and endpoint or consuming processes on the device, e.g., during IPC (Inter-Process Communication)

The degree of scrutiny that's appropriate for each state may depend on the data's importance and likelihood of being accessed. For example, data held in application memory may be more vulnerable than data on web servers to access via core dumps because attackers are more likely to gain physical access to mobile devices than to web servers.

When no data classification policy is available, use the following list of information that's generally considered sensitive:

- user authentication information (credentials, PINs, etc.)
- Personally Identifiable Information (PII) that can be abused for identity theft: social security numbers, credit card numbers, bank account numbers, health information
- device identifiers that may identify a person
- highly sensitive data whose compromise would lead to reputational harm and/or financial costs
- any data whose protection is a legal obligation
- any technical data generated by the application (or its related systems) and used to protect other data or the system itself (e.g., encryption keys).

  A definition of "sensitive data" must be decided before testing begins because detecting sensitive data leakage without a definition may be impossible.

Intelligence Gathering

Intelligence gathering involves the collection of information about the app's architecture, the business use cases the app serves, and the context in which the app operates. Such information may be classified as "environmental" or "architectural."

Environmental Information

Environmental information includes:

- The organization's goals for the app. Functionality shapes users' interaction with the app and may make some surfaces more likely than others to be targeted by attackers.
- The relevant industry. Different industries may have different risk profiles.
- Stakeholders and investors; understanding who is interested in and responsible for the app.
- Internal processes, workflows, and organizational structures. Organization-specific internal processes and workflows may create opportunities for business logic exploits.

Architectural Information

Architectural information includes:

- The mobile app: How the app accesses data and manages it in-process, how it communicates with other resources and manages user sessions, and whether it detects itself running on jailbroken or rooted phones and reacts to these situations.
- The Operating System: The operating systems and OS versions the app runs on (including Android or iOS version restrictions), whether the app is expected to run on devices that have Mobile Device Management (MDM) controls, and relevant OS vulnerabilities.
- Network: Usage of secure transport protocols (e.g., TLS), usage of strong keys and cryptographic algorithms (e.g., SHA-2) to secure network traffic encryption, usage of certificate pinning to verify the endpoint, etc.
- Remote Services: The remote services the app consumes and whether their being compromised could compromise the client.

Mapping the Application

Once the security tester has information about the app and its context, the next step is mapping the app's structure and content, e.g., identifying its entry points, features, and data.

When penetration testing is performed in a white-box or grey-box paradigm, any documents from the interior of the project (architecture diagrams, functional specifications, code, etc.) may greatly facilitate the process. If source code is available, the use of SAST tools can reveal valuable information about vulnerabilities (e.g., SQL Injection). DAST tools may support black-box testing and automatically scan the app: whereas a tester will need hours or days, a scanner may perform the same task in a few minutes. However, it's important to remember that automatic tools have limitations and will only find what they have been programmed to find. Therefore, human analysis may be necessary to augment results from automatic tools (intuition is often key to security testing).

Threat Modeling is an important artifact: documents from the workshop usually greatly support the identification of much of the information a security tester needs (entry points, assets, vulnerabilities, severity, etc.). Testers are strongly advised to discuss the availability of such documents with the client. Threat modeling should be a key part of the software development life cycle. It usually occurs in the early phases of a project.

The threat modeling guidelines defined in OWASP are generally applicable to mobile apps.

Exploitation

Unfortunately, time or financial constraints limit many pentests to application mapping via automated scanners (for vulnerability analysis, for example). Although vulnerabilities identified during the previous phase may be interesting, their relevance must be confirmed with respect to five axes:

- Damage potential - the damage that can result from exploiting the vulnerability
- Reproducibility - ease of reproducing the attack
- Exploitability - ease of executing the attack
- Affected users - the number of users affected by the attack
- Discoverability - ease of discovering the vulnerability

Against all odds, some vulnerabilities may not be exploitable and may lead to minor compromises, if any. Other vulnerabilities may seem harmless at first sight, yet be determined very dangerous under realistic test conditions. Testers who carefully go through the exploitation phase support pentesting by characterizing vulnerabilities and their effects.

Reporting

The security tester's findings will be valuable to the client only if they are clearly documented. A good pentest report should include information such as, but not limited to, the following:

- an executive summary
- a description of the scope and context (e.g., targeted systems)
- methods used
- sources of information (either provided by the client or discovered during the pentest)
- prioritized findings (e.g., vulnerabilities that have been structured by DREAD classification)
- detailed findings
- recommendations for fixing each defect

Many pentest report templates are available on the internet: Google is your friend!

Security Testing and the SDLC

Although the principles of security testing haven't fundamentally changed in recent history, software development techniques have changed dramatically. While the widespread adoption of Agile practices was speeding up software development, security testers had to become quicker and more agile while continuing to deliver trustworthy software.

The following section is focused on this evolution and describes contemporary security testing.

Security Testing during the Software Development Life Cycle

Software development is not very old, after all, so the end of developing without a framework is easy to observe. We have all experienced the need for a minimal set of rules to control work as the source code grows.

In the past, "Waterfall" methodologies were the most widely adopted: development proceeded by steps that had a predefined sequence. Limited to a single step, backtracking capability was a serious drawback of Waterfall methodologies. Although they have important positive features (providing structure, helping testers clarify where effort is needed, being clear and easy to understand, etc.), they also have negative ones (creating silos, being slow, specialized teams, etc.).

As software development matured, competition increased and developers needed to react to market changes more quickly while creating software products with smaller budgets. The idea of less structure became popular, and smaller teams collaborated, breaking silos throughout the organization. The "Agile" concept was born (Scrum, XP, and RAD are well-known examples of Agile implementations); it enabled more autonomous teams to work together more quickly.

Security wasn't originally an integral part of software development. It was an afterthought, performed at the network level by operation teams who had to compensate for poor software security! Although unintegrated security was possible when software programs were located inside a perimeter, the concept became obsolete as new kinds of software consumption emerged with web, mobile, and IoT technologies. Nowadays, security must be baked inside software because compensating for vulnerabilities is often very difficult.

> "SDLC" will be used interchangeably with "Secure SDLC" in the following section to help you internalize the idea that security is a part of software development processes. In the same spirit, we use the name DevSecOps to emphasize the fact that security is part of DevOps.

SDLC Overview

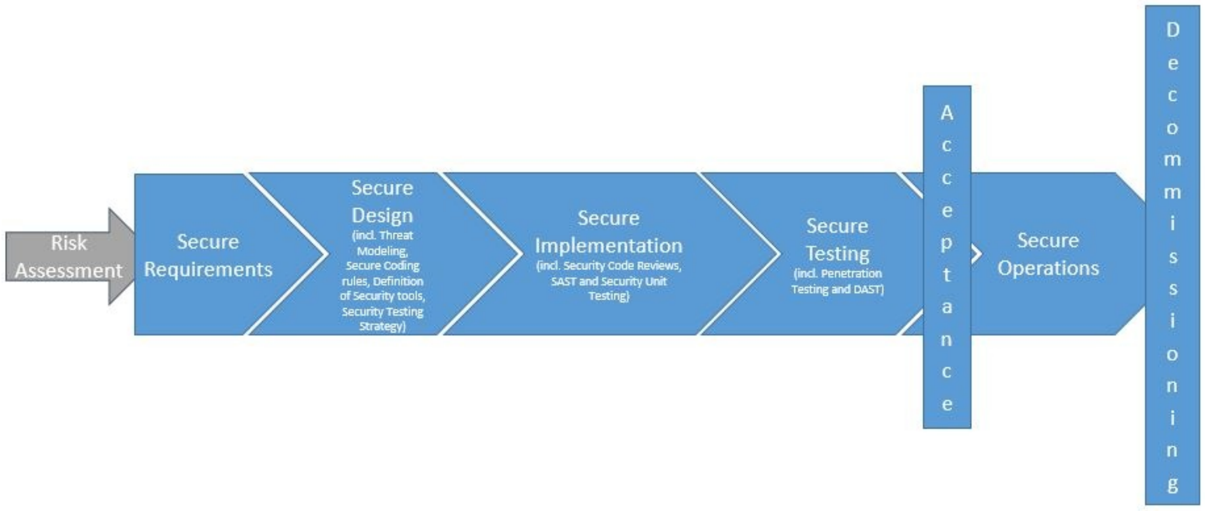
General Description of SDLC

SDLCs always consist of the same steps (the overall process is sequential in the Waterfall paradigm and iterative in the Agile paradigm):

- Perform a risk assessment for the application and its components to identify their risk profiles. These risk profiles typically depend on the organization's risk appetite and applicable regulatory requirements. The risk assessment is also based on factors, including whether the application is accessible via the internet and the kind of data the application processes and stores. All kinds of risks must be taken into account: financial, marketing, industrial, etc. Data classification policies specify which data is sensitive and how it must be secured.
- Security Requirements are determined at the beginning of a project or development cycle, when functional requirements are being gathered. Abuse Cases are added as use cases are created. Teams (including development teams) may be given security training (such as Secure Coding) if they need it. You can use the OWASP MASVS to determine the security requirements of mobile applications on the basis of the risk assessment phase. Iteratively reviewing requirements when features and data classes are added is common, especially with Agile projects.
- Threat Modeling, which is basically the identification, enumeration, prioritization, and initial handling of threats, is a foundational artifact that must be performed as architecture development and design progress. Security Architecture, a Threat Model factor, can be refined (for both software and hardware aspects) after the Threat Modeling phase. Secure Coding rules are established and the list of Security tools that will be used is created. The strategy for Security testing is clarified.
- All security requirements and design considerations should be stored in the Application Life Cycle Management (ALM) system (also known as the issue tracker) that the development/ops team uses to ensure tight integration of security requirements into the development workflow. The security requirements should contain relevant source code snippets so that developers can quickly reference the snippets. Creating a dedicated repository that's under version control and contains only these code snippets is a secure coding strategy that's more beneficial than the traditional approach (storing the guidelines in word documents or PDFs).
- Securely develop the software. To increase code security, you must complete activities such as Security Code Reviews, Static Application Security Testing, and Security Unit Testing. Although quality analogues of these security activities exist, the same logic must be applied to security, e.g., reviewing, analyzing, and testing code for security defects (for example, missing input validation, failing to free all resources, etc.).
- Next comes the long-awaited release candidate testing: both manual and automated Penetration Testing

("Pentests"). Dynamic Application Security Testing is usually performed during this phase as well.
- After the software has been Accredited during Acceptance by all stakeholders, it can be safely transitioned to Operation teams and put in Production.
- The last phase, too often neglected, is the safe Decommissioning of software after its end of use.

The picture below illustrates all the phases and artifacts:


Risk Assessment
Secure Requirements
Secure Design (incl. Threat Modeling, Secure Coding rules, Definition of Security tools, Security Testing Strategy)
Secure Implementation (incl. Security Code Reviews, SAST and Security Unit Testing)
Secure Testing (incl. Penetration Testing and DAST)
Acceptance
Secure Operations
Decommissioning

Based on the project's general risk profile, you may simplify (or even skip) some artifacts, and you may add others (formal intermediary approvals, formal documentation of certain points, etc.). Always remember two things: an SDLC is meant to reduce risks associated with software development, and it is a framework that helps you set up controls to that end. This this is a generic description of SDLC; always tailor this framework to your projects.

Defining a Test Strategy

Test strategies specify the tests that will be performed during the SDLC as well as testing frequency. Test strategies are used to make sure that the final software product meets security objectives, which are generally determined by clients' legal/marketing/corporate teams. The test strategy is usually created during the Secure Design phase, after risks have been clarified (during the Initiation phase) and before code development (the Secure Implementation phase) begins. The strategy requires input from activities such as Risk Management, previous Threat Modeling, and Security Engineering.

A Test Strategy needn't be formally written: it may be described through Stories (in Agile projects), quickly enumerated in checklists, or specified as test cases for a given tool. However, the strategy must definitely be shared because it must be implemented by a team other than the team who defined it. Moreover, all technical teams must agree to it to ensure that it doesn't place unacceptable burdens on any of them.

Test Strategies address topics such as the following:

- objectives and risk descriptions
- plans for meeting objectives, risk reduction, which tests will be mandatory, who will perform them, how and when they will be performed
- acceptance criteria

To track the testing strategy's progress and effectiveness, metrics should be defined, continually updated during the project, and periodically communicated. An entire book could be written about choosing relevant metrics; the most we can say here is that they depend on risk profiles, projects, and organizations. Examples of metrics include the following:

- the number of stories related to security controls that have been successfully implemented
- code coverage for unit tests of security controls and sensitive features

- the number of security bugs found for each build via static analysis tools
- trends in security bug backlogs (which may be sorted by urgency)

These are only suggestions; other metrics may be more relevant to your project. Metrics are powerful tools for getting a project under control, provided they give project managers a clear and synthetic perspective on what is happening and what needs to be improved.

Distinguishing between tests performed by an internal team and tests performed by an independent third party is important. Internal tests are usually useful for improving daily operations, while third-party tests are more beneficial to the whole organization. Internal tests can be performed quite often, but third-party testing happens at most once or twice a year; also, the former are less expensive than the latter. Both are necessary, and many regulations mandate tests from an independent third party because such tests can be more trustworthy.

Security Testing in Waterfall

What Waterfall Is and How Testing Activities Are Arranged

Basically, SDLC doesn't mandate the use of any development life cycle: it is safe to say that security can (and must!) be addressed in any situation.

Waterfall methodologies were popular before the 21st century. The most famous application is called the "V model," in which phases are performed in sequence and you can backtrack only a single step. The testing activities of this model occur in sequence and are performed as a whole, mostly at the point in the life cycle when most of the app development is complete. This activity sequence means that changing the architecture and other factors that were set up at the beginning of the project is hardly possible even though code may be changed after defects have been identified.

Security Testing for Agile/DevOps and DevSecOps

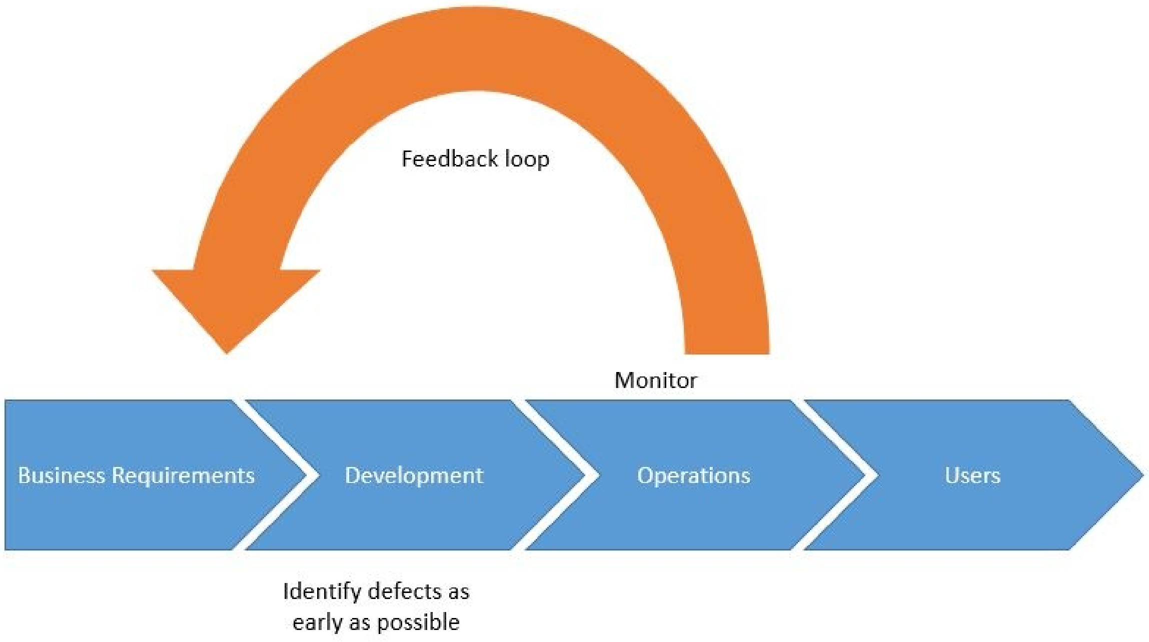
DevOps refers to practices that focus on a close collaboration between all stakeholders involved in software development (generally called Devs) and operations (generally called Ops). DevOps is not about merging Devs and Ops. Development and operations teams originally worked in silos, when pushing developed software to production could take a significant amount of time. When development teams made moving more deliveries to production necessary by working with Agile, operation teams had to speed up to match the pace. DevOps is the necessary evolution of the solution to that challenge in that it allows software to be released to users more quickly. This is largely accomplished via extensive build automation, the process of testing and releasing software, and infrastructure changes (in addition to the collaboration aspect of DevOps). This automation is embodied in the deployment pipeline with the concepts of Continuous Integration and Continuous Delivery (CI/CD).

People may assume that the term "DevOps" represents collaboration between development and operations teams only, however, as DevOps thought leader Gene Kim puts it: "At first blush, it seems as though the problems are just between dev and ops, but test is in there, and you have information security objectives, and the need to protect systems and data. These are top-level concerns of management, and they have become part of the DevOps picture."

In other words, DevOps collaboration includes quality teams, security teams, and many other teams related to the project. When you hear "DevOps" today, you should probably be thinking of something like DevOpsQATestInfoSec. Indeed, DevOps values pertain to increasing not only speed but also quality, security, reliability, stability, and resilience.

Security is just as critical to business success as the overall quality, performance, and usability of an application. As development cycles are shortened and delivery frequencies increased, making sure that quality and security are built in from the very beginning becomes essential. DevSecOps is all about adding security to DevOps processes. Most defects are identified during production. DevOps specifies best practices for identifying as many defects as possible early in the life cycle and for minimizing the number of defects in the released application.

However, DevSecOps is not just a linear process oriented towards delivering the best possible software to operations; it is also a mandate that operations closely monitor software that's in production to identify issues and fix them by forming a quick and efficient feedback loop with development. DevSecOps is a process through which Continuous Improvement is heavily emphasized.



The human aspect of this emphasis is reflected in the creation of cross-functional teams that work together to achieve business outcomes. This section is focused on necessary interactions and integrating security into the development life cycle (which starts with project inception and ends with the delivery of value to users).

What Agile and DevSecOps Are and How Testing Activities Are Arranged

Overview

Automation is a key DevSecOps practice: as stated earlier, the frequency of deliveries from development to operation increases when compared to the traditional approach, and activities that usually require time need to keep up, e.g. deliver the same added value while taking more time. Unproductive activities must consequently be abandoned, and essential tasks must be fastened. These changes impact infrastructure changes, deployment, and security:

- infrastructure is being implemented as Infrastructure as Code
- deployment is becoming more scripted, translated through the concepts of Continuous Integration and Continuous Delivery
- security activities are being automated as much as possible and taking place throughout the life cycle

The following sections provide more details about these three points.

Infrastructure as Code

Instead of manually provisioning computing resources (physical servers, virtual machines, etc.) and modifying configuration files, Infrastructure as Code is based on the use of tools and automation to fasten the provisioning process and make it more reliable and repeatable. Corresponding scripts are often stored under version control to facilitate sharing and issue resolution.

Infrastructure as Code practices facilitate collaboration between development and operations teams, with the following results:

- Devs better understand infrastructure from a familiar point of view and can prepare resources that the running application will require.

- Ops operate an environment that better suits the application, and they share a language with Devs.

Infrastructure as Code also facilitates the construction of the environments required by classical software creation projects, for development ("DEV"), integration ("INT"), testing ("PPR" for Pre-Production. Some tests are usually performed in earlier environments, and PPR tests mostly pertain to non-regression and performance with data that's similar to data used in production), and production ("PRD"). The value of infrastructure as code lies in the possible similarity between environments (they should be the same).

Infrastructure as Code is commonly used for projects that have Cloud-based resources because many vendors provide APIs that can be used for provisioning items (such as virtual machines, storage spaces, etc.) and working on configurations (e.g., modifying memory sizes or the number of CPUs used by virtual machines). These APIs provide alternatives to administrators' performing these activities from monitoring consoles.

The main tools in this domain are Puppet, Terraform, Chef and Ansible.

Deployment

The deployment pipeline's sophistication depends on the maturity of the project organization or development team. In its simplest form, the deployment pipeline consists of a commit phase. The commit phase usually involves running simple compiler checks and the unit test suite as well as creating a deployable artifact of the application. A release candidate is the latest version that has been checked into the trunk of the version control system. Release candidates are evaluated by the deployment pipeline for conformity to standards they must fulfil for deployment to production.

The commit phase is designed to provide instant feedback to developers and is therefore run on every commit to the trunk. Time constraints exist because of this frequency. The commit phase should usually be complete within five minutes, and it shouldn't take longer than ten. Adhering to this time constraint is quite challenging when it comes to security because many security tools can't be run quickly enough (#paul, #mcgraw).

CI/CD means "Continuous Integration/Continuous Delivery" in some contexts and "Continuous Integration/Continuous Deployment" in others. Actually, the logic is:

- Continuous Integration build actions (either triggered by a commit or performed regularly) use all source code to build a candidate release. Tests can then be performed and the release's compliance with security, quality, etc., rules can be checked. If case compliance is confirmed, the process can continue; otherwise, the development team must remediate the issue(s) and propose changes.
- Continuous Delivery candidate releases can proceed to the pre-production environment. If the release can then be validated (either manually or automatically), deployment can continue. If not, the project team will be notified and proper action(s) must be taken.
- Continuous Deployment releases are directly transitioned from integration to production, e.g., they become accessible to the user. However, no release should go to production if significant defects have been identified during previous activities.

The delivery and deployment of applications with low or medium sensitivity may be merged into a single step, and validation may be performed after delivery. However, keeping these two actions separate and using strong validation are strongly advised for sensitive applications.

Security

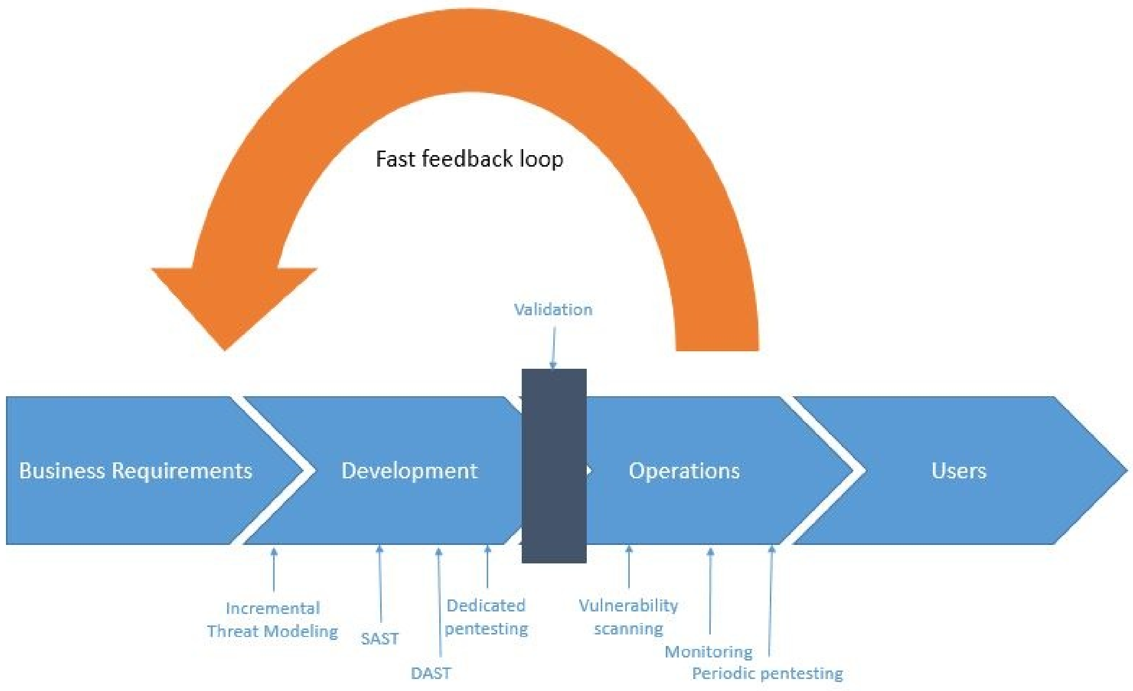
At this point, the big question is: now that other activities required for delivering code are completed significantly faster and more effectively, how can security keep up? How can we maintain an appropriate level of security? Delivering value to users more often with decreased security would definitely not be good!

Once again, the answer is automation and tooling: by implementing these two concepts throughout the project life cycle, you can maintain and improve security. The higher the expected level of security, the more controls, checkpoints, and emphasis will take place. The following are examples:

- Static Application Security Testing can take place during the development phase, and it can be integrated into the Continuous Integration process with more or less emphasis on scan results. You can establish more or less demanding Secure Coding Rules and use SAST tools to check the effectiveness of their implementation.
- Dynamic Application Security Testing may be automatically performed after the application has been built (e.g., after Continuous Integration has taken place) and before delivery, again, with more or less emphasis on results.
- You can add manual validation checkpoints between consecutive phases, for example, between delivery and deployment.

The security of an application developed with DevOps must be considered during operations. The following are examples:

- Scanning should take place regularly (at both the infrastructure and application level).
- Pentesting may take place regularly. (The version of the application used in production is the version that should be pentested, and the testing should take place in a dedicated environment and include data that's similar to the production version data. See the section on Penetration Testing for more details.)
- Active monitoring should be performed to identify issues and remediate them as soon as possible via the feedback loop.



References

- [paul] - M. Paul. Official (ISC)2 Guide to the CSSLP CBK, Second Edition ((ISC)2 Press), 2014
- [mcgraw] - G McGraw. Software Security: Building Security In, 2006

Testing Tools

To perform security testing different tools are available in order to be able to manipulate requests and responses, decompile Apps, investigate the behavior of running Apps and other test cases and automate them.

Mobile Application Security Testing Distributions

- Androl4b - A Virtual Machine For Assessing Android applications, Reverse Engineering and Malware Analysis
- Android Tamer - Android Tamer is a Debian-based Virtual/Live Platform for Android Security professionals.
- AppUse - AppUse is a Virtual Machine developed by AppSec Labs.
- Santoku - Santoku is an OS and can be run outside a VM as a standalone operating system.
- Mobile Security Toolchain - A project used to install many of the tools mentioned in this section both for Android and iOS at a machine running Mac OSX. The project installs the tools via Ansible

Static Source Code Analysis

- Checkmarx - Static Source Code Scanner that also scans source code for Android and iOS.
- Fortify - Static source code scanner that also scans source code for Android and iOS.
- Veracode - Static Analysis of iOS and Android binary

All-in-One Mobile Security Frameworks

- Appmon - AppMon is an automated framework for monitoring and tampering system API calls of native macOS, iOS and android apps.
- Mobile Security Framework - MobSF - Mobile Security Framework is an intelligent, all-in-one open source mobile application (Android/iOS) automated pen-testing framework capable of performing static and dynamic analysis.
- Needle - Needle is an open source, modular framework to streamline the process of conducting security assessments of iOS apps including Binary Analysis, Static Code Analysis, Runtime Manipulation using Cycript and Frida hooking, and so on.
- objection - objection is a runtime mobile security assessment framework that does not require a jailbroken or rooted device for both iOS and Android, due to the usage of Frida.

Tools for Android

Reverse Engineering and Static Analysis

- Androguard - Androguard is a python based tool, which can use to disassemble and decompile android apps.
- Android Debug Bridge - adb - Android Debug Bridge (adbis a versatile command line tool that lets you communicate with an emulator instance or connected Android device.
- APKInspector - APKinspector is a powerful GUI tool for analysts to analyze the Android applications.
- APKTool - A tool for reverse engineering 3rd party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making some modifications.
- android-classyshark - ClassyShark is a standalone binary inspection tool for Android developers.
- Sign - Sign.jar automatically signs an apk with the Android test certificate.
- Jadx - Dex to Java decompiler: Command line and GUI tools for produce Java source code from Android Dex and Apk files.
- Oat2dex - A tool for converting .oat file to .dex files.
- FindBugs - Static Analysis tool for Java
- FindSecurityBugs - FindSecurityBugs is a extension for FindBugs which include security rules for Java

applications.

- Qark - This tool is designed to look for several security related Android application vulnerabilities, either in source code or packaged APKs.
- SUPER - SUPER is a command-line application that can be used in Windows, MacOS X and Linux, that analyzes .apk files in search for vulnerabilities. It does this by decompressing APKs and applying a series of rules to detect those vulnerabilities.
- AndroBugs - AndroBugs Framework is an efficient Android vulnerability scanner that helps developers or hackers find potential security vulnerabilities in Android applications. No need to install on Windows.
- Simplify - A tool for de-obfuscating android package into Classes.dex which can be use Dex2jar and JD-GUI to extract contents of dex file.
- ClassNameDeobfuscator - Simple script to parse through the .smali files produced by apktool and extract the .source annotation lines.
- Android backup extractor - Utility to extract and repack Android backups created with adb backup (ICS+). Largely based on BackupManagerService.java from AOSP.
- VisualCodeGrepper - Static Code Analysis Tool for several programming languages including Java
- ByteCodeViewer - Five different Java Decompiles, Two Bytecode Editors, A Java Compiler, Plugins, Searching, Supports Loading from Classes, JARs, Android APKs and More.

Dynamic and Runtime Analysis

- Cydia Substrate - Cydia Substrate for Android enables developers to make changes to existing software with Substrate extensions that are injected in to the target process's memory.
- Xposed Framework - Xposed framework enables you to modify the system or application aspect and behavior at runtime, without modifying any Android application package(APKor re-flashing.
- logcat-color - A colorful and highly configurable alternative to the adb logcat command from the Android SDK.
- Inspeckage - Inspeckage is a tool developed to offer dynamic analysis of Android applications. By applying hooks to functions of the Android API, Inspeckage will help you understand what an Android application is doing at runtime.
- Frida - The toolkit works using a client-server model and lets you inject in to running processes not just on Android, but also on iOS, Windows and Mac.
- Diff-GUI - A Web framework to start instrumenting with the avaliable modules, hooking on native, inject JavaScript using Frida.
- AndBug - AndBug is a debugger targeting the Android platform's Dalvik virtual machine intended for reverse engineers and developers.
- Cydia Substrate: Introspy-Android - Blackbox tool to help understand what an Android application is doing at runtime and assist in the identification of potential security issues.
- Drozer - Drozer allows you to search for security vulnerabilities in apps and devices by assuming the role of an app and interacting with the Dalvik VM, other apps' IPC endpoints and the underlying OS.
- VirtualHook - VirtualHook is a hooking tool for applications on Android ART(>=5.0). It's based on VirtualApp and therefore does not require root permission to inject hooks.

Bypassing Root Detection and Certificate Pinning

- Xposed Module: Just Trust Me - Xposed Module to bypass SSL certificate pinning.
- Xposed Module: SSLUnpinning - Android Xposed Module to bypass SSL certificate validation (Certificate Pinning)).
- Cydia Substrate Module: Android SSL Trust Killer - Blackbox tool to bypass SSL certificate pinning for most applications running on a device.
- Cydia Substrate Module: RootCoak Plus - Patch root checking for commonly known indications of root.
- Android-ssl-bypass - an Android debugging tool that can be used for bypassing SSL, even when certificate pinning is implemented, as well as other debugging tasks. The tool runs as an interactive console.

Tools for iOS

Access Filesystem on iDevice

- FileZilla - It supports FTP, SFTP, and FTPS (FTP over SSL/TLS).
- Cyberduck - Libre FTP, SFTP, WebDAV, S3, Azure & OpenStack Swift browser for Mac and Windows.
- itunnel - Use to forward SSH via USB.
- iFunbox - The File and App Management Tool for iPhone, iPad & iPod Touch.

Reverse Engineering and Static Analysis

- otool - The otool command displays specified parts of object files or libraries.
- Clutch - Decrypted the application and dump specified bundleID into binary or .ipa file.
- Dumpdecrypted - Dumps decrypted mach-o files from encrypted iPhone applications from memory to disk. This tool is necessary for security researchers to be able to look under the hood of encryption.
- class-dump - A command-line utility for examining the Objective-C runtime information stored in Mach-O files.
- Flex2 - Flex gives you the power to modify apps and change their behavior.
- Weak Classdump - A Cycript script that generates a header file for the class passed to the function. Most useful when you cannot classdump or dumpdecrypted , when binaries are encrypted etc.
- IDA Pro - IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and debugger that offers so many features it is hard to describe them all.
- HopperApp - Hopper is a reverse engineering tool for OS X and Linux, that lets you disassemble, decompile and debug your 32/64bits Intel Mac, Linux, Windows and iOS executables.
- Radare2 - Radare2 is a unix-like reverse engineering framework and command line tools.
- iRET - The iOS Reverse Engineering Toolkit is a toolkit designed to automate many of the common tasks associated with iOS penetration testing.
- Plutil - plutil is a program that can convert .plist files between a binary version and an XML version.

Dynamic and Runtime Analysis

- cycript - Cycript allows developers to explore and modify running applications on either iOS or Mac OS X using a hybrid of Objective-C++ and JavaScript syntax through an interactive console that features syntax highlighting and tab completion.
- iNalyzer - AppSec Labs iNalyzer is a framework for manipulating iOS applications, tampering with parameters and method.
- idb - idb is a tool to simplify some common tasks for iOS pentesting and research.
- snoop-it - A tool to assist security assessments and dynamic analysis of iOS Apps.
- Introspy-iOS - Blackbox tool to help understand what an iOS application is doing at runtime and assist in the identification of potential security issues.
- gdb - A tool to perform runtime analysis of IOS applications.
- lldb - LLDB debugger by Apple’s Xcode is used for debugging iOS applications.
- keychaindumper - A tool to check which keychain items are available to an attacker once an iOS device has been jailbroken.
- BinaryCookieReader - A tool to dump all the cookies from the binary Cookies.binarycookies file.
- Burp Suite Mobile Assistant - A tool to bypass certificate pinning and is able to inject into apps.

Bypassing Root Detection and SSL Pinning

- SSL Kill Switch 2 - Blackbox tool to disable SSL certificate validation - including certificate pinning - within iOS and OS X Apps.
- iOS TrustMe - Disable certificate trust checks on iOS devices.

- Xcon - A tool for bypassing Jailbreak detection.
- tsProtector - Another tool for bypassing Jailbreak detection.

Tools for Network Interception and Monitoring

- Tcpdump - A command line packet capture utility.
- Wireshark - An open-source packet analyzer.
- Canape - A network testing tool for arbitrary protocols.
- Mallory - A Man in The Middle Tool (MiTM)) that is used to monitor and manipulate traffic on mobile devices and applications.

Interception Proxies

- Burp Suite - Burp Suite is an integrated platform for performing security testing of applications.
- OWASP ZAP - The OWASP Zed Attack Proxy (ZAPis a free security tools which can help you automatically find security vulnerabilities in your web applications and web services.
- Fiddler - Fiddler is an HTTP debugging proxy server application which can captures HTTP and HTTPS traffic and logs it for the user to review. Fiddler can also be used to modify HTTP traffic for troubleshooting purposes as it is being sent or received.
- Charles Proxy - HTTP proxy / HTTP monitor / Reverse Proxy that enables a developer to view all of the HTTP and SSL / HTTPS traffic between their machine and the Internet.

IDEs

- Android Studio - is the official integrated development environment (IDE) for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development.
- IntelliJ - IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software.
- Eclipse - Eclipse is an integrated development environment (IDE) used in computer programming, and is the most widely used Java IDE.
- Xcode - Xcode is an integrated development environment (IDE) available only for macOS to create apps for iOS, watchOS, tvOS and macOS.

Vulnerable applications

The applications listed below can be used as training materials.

Android

- DVHMA - is an hybrid mobile app (for Android) that intentionally contains vulnerabilities.
- Crackmes - is a set of applications to test your Android application hacking skills.
- OMTG Android app - is a vulnerable Android application with vulnerabilities similar to the testcases described in this document.
- Digitalbank - a vulnerable app which comes from 2015, can be used on older Android platforms. Note: this is not tested by the authors.
- DIVA Android - is an App intentionally designed to be insecure which has received updates in 2016 and contains 13 different challenges. Note: this is not tested by the authors.
- InsecureBankv2 - is a vulnerable Android application is named "InsecureBankv2" and is made for security enthusiasts and developers to learn the Android insecurities by testing this vulnerable application. It has been updated in 2018 and contains over 25 vulnerabilities.
- DodoVulnerableBank - An insecure Android app from 2015. Note: this is not tested by the authors.

iOS

- Crackmes - is a set of applications to test your iOS application hacking skills.
- Myriam - A vulnerable iOS app with iOS security challenges.
- DVIA - A vulnerable iOS app, written in objective-C with a set of vulnerabilities. Additional lessons can be found at the projects website.
- DVIA-V2 - A vulnerable iOS app, written in Swift with over 15 vulnerabilities.

Suggested Reading

Mobile App Security

Android

- Dominic Chell, Tyrone Erasmus, Shaun Colley, Ollie Whitehous (2015) *Mobile Application Hacker's Handbook*. Wiley. Available at: http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118958500.html
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva, Stephen A. Ridley, Georg Wicherski (2014) *Android Hacker's Handbook*. Wiley. Available at: http://www.wiley.com/WileyCDA/WileyTitle/productCd-111860864X.html
- Godfrey Nolan (2014) *Bulletproof Android*. Addison-Wesley Professional. Available at: https://www.amazon.com/Bulletproof-Android-Practical-Building-Developers/dp/0133993329
- Nikolay Elenkov (2014) *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press. Available at: https://nostarch.com/androidsecurity

iOS

- Charlie Miller, Dionysus Blazakis, Dino Dai Zovi, Stefan Esser, Vincenzo Iozzo, Ralf-Philipp Weinmann (2012) *iOS Hacker's Handbook*. Wiley. Available at: http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118204123.html
- David Thiel (2016) *iOS Application Security, The Definitive Guide for Hackers and Developers*. no starch press. Available at: https://www.nostarch.com/iossecurity
- Jonathan Levin (2017), *Mac OS X and iOS Internals*, Wiley. Available at: http://newosxbook.com/index.php

Misc

Reverse Engineering

- Bruce Dang, Alexandre Gazet, Elias Backaalany (2014) *Practical Reverse Engineering*. Wiley. Available at: http://as.wiley.com/WileyCDA/WileyTitle/productCd-1118787315,subjectCd-CSJ0.html
- Skakenunny, Hangcom *iOS App Reverse Engineering*. Online. Available at: https://github.com/iosre/iOSAppReverseEngineering/
- Bernhard Mueller (2016) *Hacking Soft Tokens - Advanced Reverse Engineering on Android*. HITB GSEC Singapore. Available at: http://gsec.hitb.org/materials/sg2016/D1%20-%20Bernhard%20Mueller%20-%20Attacking%20Software%20Tokens.pdf
- Dennis Yurichev (2016) *Reverse Engineering for Beginners*. Online. Available at: https://github.com/dennis714/RE-for-beginners
- Michael Hale Ligh, Andrew Case, Jamie Levy, Aaron Walters (2014) *The Art of Memory Forensics.* Wiley. Available at: http://as.wiley.com/WileyCDA/WileyTitle/productCd-1118825098.html