NIST Special Publication 800-163

Vetting the Security of Mobile Applications

Steve Quirolgico
Jeffrey Voas
Tom Karygiannis
Christoph Michael
Karen Scarfone

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-163

C O M P U T E R   S E C U R I T Y

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Special Publication 800-163

Vetting the Security of Mobile Applications

Steve Quirolgico, Jeffrey Voas, Tom Karygiannis
Computer Security Division
Information Technology Laboratory

Christoph Michael
Leidos
Reston, VA

Karen Scarfone
Scarfone Cybersecurity
Clifton, VA

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-163

January 2015

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Acting Under Secretary of Commerce for Standards and Technology and Acting Director

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Management Act of 2002 (FISMA), 44 U.S.C. § 3541 *et seq.*, Public Law 107-347. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), *Securing Agency Information Systems*, as analyzed in Circular A-130, Appendix IV: *Analysis of Key Sections*. Supplemental information is provided in Circular A-130, Appendix III, *Security of Federal Automated Information Resources*.

National Institute of Standards and Technology Special Publication 800-163
Natl. Inst. Stand. Technol. Spec. Publ. 800-163, 44 pages (January 2015)
CODEN: NSPUE2

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-163

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
nist800-163@nist.gov

Reports on Computer Systems Technology

Abstract

The purpose of this document is to help organizations (1) understand the process for vetting the security of mobile applications, (2) plan for the implementation of an app vetting process, (3) develop app security requirements, (4) understand the types of app vulnerabilities and the testing methods used to detect those vulnerabilities, and (5) determine if an app is acceptable for deployment on the organization's mobile devices.

Keywords

app vetting; apps; malware; mobile devices; security requirements; security vetting; smartphones; software assurance; software security; software testing; software vetting

Acknowledgments

Special thanks to the National Security Agency's Center for Assured Software for providing Appendices B and C which describe Android and iOS app vulnerabilities. Also, thanks to all the organizations and individuals who made contributions to the publication including Robert Martin of The MITRE Corporation, Paul Black and Irena Bojanova of NIST, the Department of Homeland Security (DHS), and the Department of Justice (DOJ). Finally, special thanks to the Defense Advanced Research Projects Agency (DARPA) Transformative Applications (TransApps) program for funding NIST research in mobile app security vetting.

Trademarks

Table of Contents

1 Introduction 1

1.1 Traditional vs. Mobile Application Security Issues 1

1.2 App Vetting Basics 2

1.3 Purpose and Scope 3

1.4 Audience 3

1.5 Document Structure 4

2 App Vetting Process 5

2.1 App Testing 5

2.2 App Approval/Rejection 5

2.3 Planning 6

2.3.1 Developing Security Requirements 6

2.3.2 Understanding Vetting Limitations 9

2.3.3 Budget and Staffing 10

3 App Testing 11

3.1 General Requirements 11

3.1.1 Enabling Authorized Functionality 12

3.1.2 Preventing Unauthorized Functionality 12

3.1.3 Limiting Permissions 13

3.1.4 Protecting Sensitive Data 14

3.1.5 Securing App Code Dependencies 15

3.1.6 Testing App Updates 16

3.2 Testing Approaches 16

3.2.1 Correctness Testing 16

3.2.2 Source Code Versus Binary Code 17

3.2.3 Static Versus Dynamic Analysis 17

3.2.4 Automated Tools 18

3.3 Sharing Results 20

4 App Approval/Rejection 21

4.1 Report and Risk Auditing 21

4.2 Organization-Specific Vetting Criteria 22

4.3 Final Approval/Rejection 23

Appendix A— Recommendations 24

A.1 Planning 24

A.2 App Testing 24

A.3 App Approval/Rejection 25

Appendix B— Android App Vulnerability Types 26

Appendix C— iOS App Vulnerability Types 29

Appendix D— Glossary 32

Appendix E— Acronyms and Abbreviations 33

Appendix F— References 34

List of Figures and Tables

Figure 1. An app vetting process and its related actors. 6

Figure 2. Organizational app security requirements. 8

Table 1. Android Vulnerabilities, A Level 26

Table 2. Android Vulnerabilities by Level 27

Table 3. iOS Vulnerability Descriptions, A Level 29

Table 4. iOS Vulnerabilities by Level 30

Executive Summary

Recently, organizations have begun to deploy mobile applications (or *apps*) to facilitate their business processes. Such apps have increased productivity by providing an unprecedented level of connectivity between employees, vendors, and customers, real-time information sharing, unrestricted mobility, and improved functionality. Despite the benefits of mobile apps, however, the use of apps can potentially lead to serious security issues. This is so because, like traditional enterprise applications, apps may contain software vulnerabilities that are susceptible to attack. Such vulnerabilities may be exploited by an attacker to gain unauthorized access to an organization's information technology resources or the user's personal data.

To help mitigate the risks associated with app vulnerabilities, organizations should develop security requirements that specify, for example, how data used by an app should be secured, the environment in which an app will be deployed, and the acceptable level of risk for an app. To help ensure that an app conforms to such requirements, a process for evaluating the security of apps should be performed. We refer to this process as an *app vetting process*. An app vetting process is a sequence of activities that aims to determine if an app conforms to an organization's security requirements. An app vetting process comprises two main activities: *app testing* and *app approval/rejection*. The app testing activity involves the testing of an app for software vulnerabilities by services, tools, and humans to derive vulnerability reports and risk assessments. The app approval/rejection activity involves the evaluation of these reports and risk assessments, along with additional criteria, to determine the app's conformance with organizational security requirements and ultimately, the approval or rejection of the app for deployment on the organization's mobile devices.

Before using an app vetting process, an organization must first plan for its implementation. To facilitate the planning of an app vetting process implementation, this document:

- describes the general and context-sensitive requirements that make up an organization's app security requirements.
- provides a questionnaire for identifying the security needs and expectations of an organization that are necessary to develop the organization's app security requirements.

With respect to the app testing activity of an app vetting process, this document describes:

- the app testing activity and its related actors.
- issues and recommendations surrounding the security of app files during testing.
- general app security requirements including: preventing unauthorized functionality, protecting sensitive data, and securing app code dependencies.
- issues and recommendations surrounding the use of source code vs. binary code, static vs. dynamic analysis, and automated tools for testing apps.
- issues and recommendations surrounding the sharing of test results.
- Android and iOS app vulnerabilities.

With respect to the app approval/rejection activity of an app vetting process, this document describes:

- the app approval/rejection activity and its related actors.
- organization-specific vetting criteria that may be used to help determine an app's conformance to context-sensitive security requirements.

1 Introduction

When deploying a new technology, an organization should be aware of the potential security impact it may have on the organization's IT resources, data, and users. While new technologies may offer the promise of productivity gains and new capabilities, they may also present new risks. Thus, it is important for an organization's IT professionals and users to be fully aware of these risks and either develop plans to mitigate them or accept their consequences.

Recently, there has been a paradigm shift where organizations have begun to deploy new mobile technologies to facilitate their business processes. Such technologies have increased productivity by providing (1) an unprecedented level of connectivity between employees, vendors, and customers; (2) real-time information sharing; (3) unrestricted mobility; and (4) improved functionality. These mobile technologies comprise mobile devices (e.g., smartphones and tablets) and related mobile applications (or *apps*) that provide mission-specific capabilities needed by users to perform their duties within the organization (e.g., sales, distribution, and marketing). Despite the benefits of mobile apps, however, the use of apps can potentially lead to serious security risks. This is so because, like traditional enterprise applications, apps may contain software vulnerabilities that are susceptible to attack. Such vulnerabilities may be exploited by an attacker to steal information or control a user's device.

To help mitigate the risks associated with software vulnerabilities, organizations should employ software assurance processes. Software assurance refers to "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended manner" [1]. The software assurance process includes "the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures" [2]. A number of government and industry legacy software assurance standards exist that are primarily directed at the process for developing applications that require a high level of assurance (e.g., space flight, automotive systems, and critical defense systems).[1] Although considerable progress has been made in the past decades in the area of software assurance, and research and development efforts have resulted in a growing market of software assurance tools and services, the state of practice for many today still includes manual activities that are time-consuming, costly, and difficult to quantify and make repeatable. The advent of mobile computing adds new challenges because it does not necessarily support traditional software assurance techniques.

1.1 Traditional vs. Mobile Application Security Issues

The economic model of the rapidly evolving app marketplace challenges the traditional software development process. App developers are attracted by the opportunities to reach a market of millions of users very quickly. However, such developers may have little experience building quality software that is secure and do not have the budgetary resources or motivation to conduct extensive testing. Rather than performing comprehensive software tests on their code before making it available to the public, developers often release apps that contain functionality flaws and/or security-relevant weaknesses. That can leave an app, the user's device, and the user's network vulnerable to exploitation by attackers. Developers and users of these apps often tolerate buggy, unreliable, and insecure code in exchange for the low cost. In addition, app developers typically update their apps much more frequently than traditional applications.

[1] Examples of these software assurance standards include DO-178B, *Software Considerations in Airborne Systems and Equipment Certification* [3], IEC 61508 *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related System* [4], and ISO 26262 *Road vehicles -- Functional safety* [5].

Organizations that once spent considerable resources to develop in-house applications are taking advantage of inexpensive third-party apps and web services to improve their organization's productivity. Increasingly, business processes are conducted on mobile devices. This contrasts with the traditional information infrastructure where the average employee uses only a handful of applications and web-based enterprise databases. Mobile devices provide access to potentially millions of apps for a user to choose from. This trend challenges the traditional mechanisms of enterprise IT security software where software exists within a tightly controlled environment and is uniform throughout the organization.

Another major difference between apps and enterprise applications is that unlike a desktop computing system, far more precise and continuous device location information, physical sensor data, personal health metrics, and pictures and audio about a user can be exposed through apps. Apps can sense and store information including user personally identifiable information (PII). Although many apps are advertised as being free to consumers, the hidden cost of these apps may be selling the user's profile to marketing companies or online advertising agencies.

There are also differences between the network capabilities of traditional computers that enterprise applications run on and mobile devices that apps run on. Unlike traditional computers that connect to the network via Ethernet, mobile devices have access to a wide variety of network services including Wireless Fidelity (Wi-Fi), 2G/3G, and 4G/Long Term Evolution (LTE). This is in addition to short-range data connectivity provided by services such as Bluetooth and Near Field Communications (NFC). All of these mechanisms of data transmission are typically available to apps that run on a mobile device and can be vectors for remote exploits. Further, mobile devices are not physically protected to the same extent as desktop or laptop computers and can therefore allow an attacker to more easily (physically) acquire a lost or stolen device.

1.2 App Vetting Basics

To provide software assurance for apps, organizations should develop security requirements that specify, for example, how data used by an app should be secured, the environment in which an app will be deployed, and the acceptable level of risk for an app (see [6] for more information). To help ensure that an app conforms to such requirements, a process for evaluating the security of apps should be performed. We refer to this process as an *app vetting process*. An app vetting process is a sequence of activities that aims to determine if an app conforms to the organization's security requirements.[2] This process is performed on an app after the app has been developed and released for distribution but prior to its deployment on an organization's mobile device. Thus, an app vetting process is distinguished from software assurance processes that may occur during the software development life cycle of an app. Note that an app vetting process typically involves analysis of an app's compiled, binary representation but can also involve analysis of the app's source code if it is available.

The software distribution model that has arisen with mobile computing presents a number of software assurance challenges, but it also creates opportunities. An app vetting process acknowledges the concept that someone other than the software vendor is entitled to evaluate the software's behavior, allowing organizations to evaluate software in the context of their own security policies, planned use, and risk tolerance. However, distancing a developer from an organization's app vetting process can also make those activities less effective with respect to improving the secure behavior of the app.

---

[2] The app vetting process can also be used to assess other app issues including reliability, performance, and accessibility but is primarily intended to assess security-related issues.

App stores may perform app vetting processes to verify compliance with their own requirements. However, because each app store has its own unique, and not always transparent, requirements and vetting processes, it is necessary to consult current agreements and documentation for a particular app store to assess its practices. Organizations should not assume that an app has been fully vetted and conforms to their security requirements simply because it is available through an official app store. Third-party assessments that carry a moniker of "approved by" or "certified by" without providing details of which tests are performed, what the findings were, or how apps are scored or rated, do not provide a reliable indication of software assurance. These assessments are also unlikely to take organization-specific requirements and recommendations into account, such as federal-specific cryptography requirements.

Although a "one size fits all" approach to app vetting is not plausible, the basic findings from one app vetting effort may be reusable by others. Leveraging another organization's findings for an app should be contemplated to avoid duplicating work and wasting scarce app vetting resources. With appropriate standards for scoping, managing, licensing, and recording the findings from software assurance activities in consistent ways, app vetting results can be looked at collectively, and common problems and solutions may be applicable across the industry or with similar organizations.

An app vetting process should be included as part of the organization's overall security strategy. If the organization has a formal process for establishing and documenting security requirements, an app vetting process should be able to import those requirements created by the process. Bug and incident reports created by the organization could also be imported by an app vetting process, as could public advisories concerning vulnerabilities in commercial and open source apps and libraries, and conversely, results from app vetting processes may get stored in the organization's bug tracking system.

1.3 Purpose and Scope

The purpose of this document is to help organizations (1) understand the app vetting process, (2) plan for the implementation of an app vetting process, (3) develop app security requirements, (4) understand the types of app vulnerabilities and the testing methods used to detect those vulnerabilities, and (5) determine if an app is acceptable for deployment on the organization's mobile devices. Ultimately, the acceptance of an app depends on the organization's security requirements which, in turn, depend on the environment in which the app is deployed, the context in which it will be used, and the underlying mobile technologies used to run the app. This document highlights those elements that are particularly important to be considered before apps are approved as "fit-for-use."

Note that this document does not address the security of the underlying mobile platform and operating system, which are addressed in other publications [6, 7, 8]. While these are important characteristics for organizations to understand and consider in selecting mobile devices, this document is focused on how to vet apps after the choice of platform has been made. Similarly, discussion surrounding the security of web services used to support back-end processing for apps is out of scope for this document.

1.4 Audience

This document is intended for organizations that plan to implement an app vetting process or leverage existing app vetting results from other organizations. It is also intended for developers that are interested in understanding the types of software vulnerabilities that may arise in their apps during the app's software development life cycle.

1.5 Document Structure

The remainder of this document is organized into the following sections and appendices:

- Section 2 presents an overview of an app vetting process including recommendations for planning the implementation of an app vetting process.
- Section 3 describes the app testing activity of an app vetting process. Organizations interested in leveraging only existing security reports and risk assessments from other organizations can skip this section.
- Section 4 describes the app approval/rejection activity of an app vetting process.
- Appendix A describes recommendations related to the app testing and app approval/rejection activities of an app vetting process as well as the planning of an app vetting process implementation.
- Appendices B and C identify and define platform-specific vulnerabilities for apps running on the Android and iOS operating systems, respectively.
- Appendix D defines selected terms used in this document.
- Appendix E defines selected acronyms and abbreviations used in this document.
- Appendix F lists references used in this document.

2 App Vetting Process

An app vetting process comprises a sequence of two main activities: *app testing* and *app approval/rejection*. In this section, we provide an overview of these two activities as well as provide recommendations for planning the implementation of an app vetting process.

2.1 App Testing

An app vetting process begins when an app is submitted by an *administrator* to one or more *analyzers* for testing. An administrator is a member of the organization who is responsible for deploying, maintaining, and securing the organization's mobile devices as well as ensuring that deployed devices and their installed apps conform to the organization's security requirements. Apps that are submitted by an administrator for testing will typically be acquired from an app store or an app developer, each of which may be internal or external to the organization. Note that the activities surrounding an administrator's acquisition of an app are not part of an app vetting process.

An analyzer is a service, tool, or human that tests an app for specific software vulnerabilities and may be internal or external to the organization. When an app is received by an analyzer, the analyzer may perform some preprocessing of the app prior to testing the app. Preprocessing of an app may be used to determine the app's suitability for testing and may involve ensuring that the app decompiles correctly, extracting meta-data from the app (e.g., app name and version number) and storing the app in a local database.

After an app has been received and preprocessed by an analyzer, the analyzer then tests the app for the presence of software vulnerabilities. Such testing may include a wide variety of tests including static and dynamic analyses and may be performed in an automated or manual fashion. Note that the tests performed by an analyzer are not organization-specific but instead are aimed at identifying software vulnerabilities that may be common across different apps. After testing an app, an analyzer generates a report that identifies detected software vulnerabilities. In addition, the analyzer generates a risk assessment that estimates the likelihood that a detected vulnerability will be exploited and the impact that the detected vulnerability may have on the app or its related device or network. Risk assessments are typically represented as ordinal values indicating the severity of the risk (e.g., low-, moderate-, and high-risk).

2.2 App Approval/Rejection

After the report and risk assessment are generated by an analyzer, they are made available to one or more *auditors* of the organization. An auditor is a member of the organization who inspects reports and risk assessments from one or more analyzers to ensure that an app meets the security requirements of the organization. The auditor also evaluates additional criteria to determine if the app violates any organization-specific security requirements that could not be ascertained by the analyzers. After evaluating all reports, risk assessments, and additional criteria, the auditor then collates this information into a single report and risk assessment and derives a recommendation for approving or rejecting the app based on the overall security posture of the app. This recommendation is then made available to an *approver*. An approver is a high-level member of the organization responsible for determining which apps will be deployed on the organization's mobile devices. An approver uses the recommendations provided by one or more auditors that describe the security posture of the app as well as other non-security-related criteria to determine the organization's official approval or rejection of an app. If an app is approved by the approver, the administrator is then permitted to deploy the app on the organization's mobile devices. If, however, the app is rejected, the organization will follow specified procedures for identifying a suitable alternative app or rectifying issues with the problematic app. Figure 1 shows an app vetting process and its related actors.

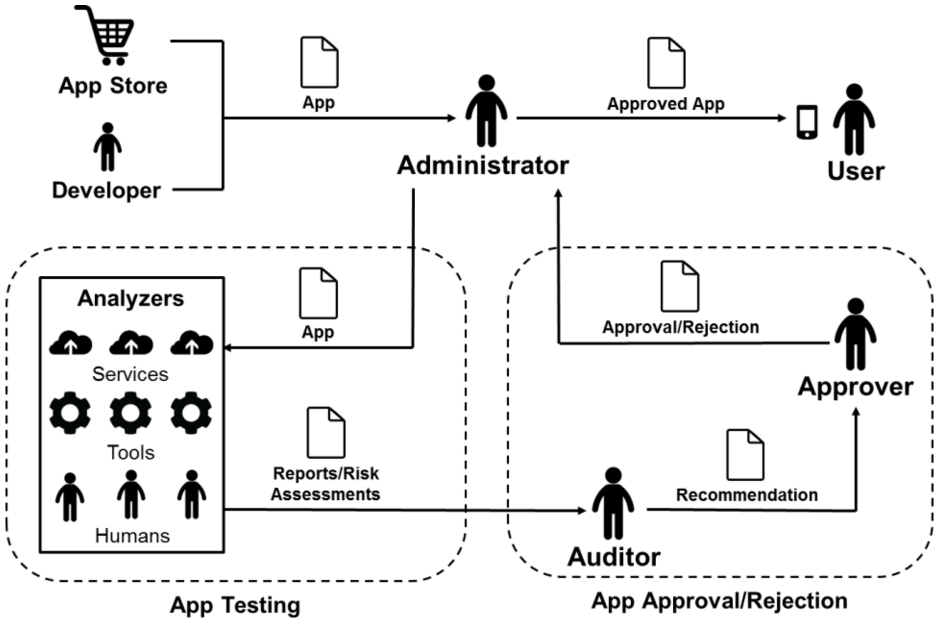

Figure 1. An app vetting process and its related actors.

Note that an app vetting process can be performed manually or through systems that provide semi-automated management of the app testing and app approval/rejection activities [9]. Further note that although app vetting processes may vary among organizations, organizations should strive to implement a process that is repeatable, efficient, consistent, and that limits errors (e.g., false positive and false negative results).

2.3 Planning

Before an organization can implement an app vetting process, it is necessary for the organization to first (1) develop app security requirements, (2) understand the limitations of app vetting, and (3) procure a budget and staff for supporting the app vetting process. Recommendations for planning the implementation of an app vetting process are described in Appendix A.1.

2.3.1 Developing Security Requirements

App security requirements state an organization's expectations for app security and drive the evaluation process. When possible, tailoring app security assessments to the organization's unique mission requirements, policies, risk tolerance, and available security countermeasures is more cost-effective in time and resources. Such assessments could minimize the risk of attackers exploiting behaviors not taken into account during vetting, since foreseeing all future insecure behaviors during deployment is unrealistic. Unfortunately, it is not always possible to get security requirements from app end users; the user base may be too broad, may not be able to state their security requirements concretely enough for testing, or may not have articulated any security requirements or expectations of secure behavior.

An organization's app security requirements comprise two types of requirements: *general* and *context-sensitive*. A general requirement is an app security requirement that specifies a software characteristic or behavior that an app should exhibit in order to be considered secure. For an app, the satisfaction or violation of a general requirement is determined by analyzers that test the app for software vulnerabilities during the app testing activity of an app vetting process. If an analyzer detects a software vulnerability in an app, the app is considered to be in violation of a general requirement. Examples of general requirements include *Apps must prevent unauthorized functionality* and *Apps must protect sensitive data*. We describe general requirements in Section 3.1.

A context-sensitive requirement is an app security requirement that specifies how apps should be used by the organization to ensure the organization's security posture. For an app, the satisfaction or violation of a context-sensitive requirement is not based on the presence or absence of a software vulnerability and thus cannot be determined by analyzers, but instead must be determined by an auditor who uses organization-specific vetting criteria for the app during the app approval/rejection activity of an app vetting process. Such criteria may include the app's intended set of users or intended deployment environment. If an auditor determines that the organization-specific vetting criteria for an app conflicts with a context-sensitive requirement, the app is considered to be in violation of the context-sensitive requirement. Examples of context-sensitive requirements include *Apps that access a network must not be used in a sensitive compartmented information facility (SCIF)* and *Apps that record audio or video must only be used by classified personnel*. We describe organization-specific vetting criteria in Section 4.2. The relationship between the organization's app security requirements and general and context-sensitive requirements is shown in Figure 2.

Here, an organization's app security requirements comprise a subset of general requirements and a subset of context-sensitive requirements. The shaded areas represent an organization's app security requirements that are applied to one or more apps. During the app approval/rejection activity of an app vetting process, the auditor reviews reports and risk assessments generated by analyzers to determine an app's satisfaction or violation of general requirements as well as evaluates organization-specific vetting criteria to determine the app's satisfaction or violation of context-sensitive requirements.

Developing app security requirements involves identifying the security needs and expectations of the organization and identifying both general and context-sensitive requirements that address those needs and expectations. For example, if an organization has a need to ensure that apps do not leak PII, the general requirement *Apps must not leak PII* should be defined. If the organization has an additional need to ensure that apps that record audio or video must not be used in a SCIF, then the context-sensitive requirement *Apps that record audio or video must not be used in a SCIF* should be used. After deriving general requirements, an organization should explore available analyzers that test for the satisfaction or violation of those requirements. In this case, for example, an analyzer that tests an app for the leakage of PII should be used. Similarly, for context-sensitive requirements, an auditor should identify appropriate organization-specific vetting criteria (e.g., the features or purpose of the app and the intended deployment environment) to determine if the app records audio or video and if the app will be used in a SCIF.

The following questionnaire may be used by organizations to help identify their specific security needs in order to develop their app security requirements.

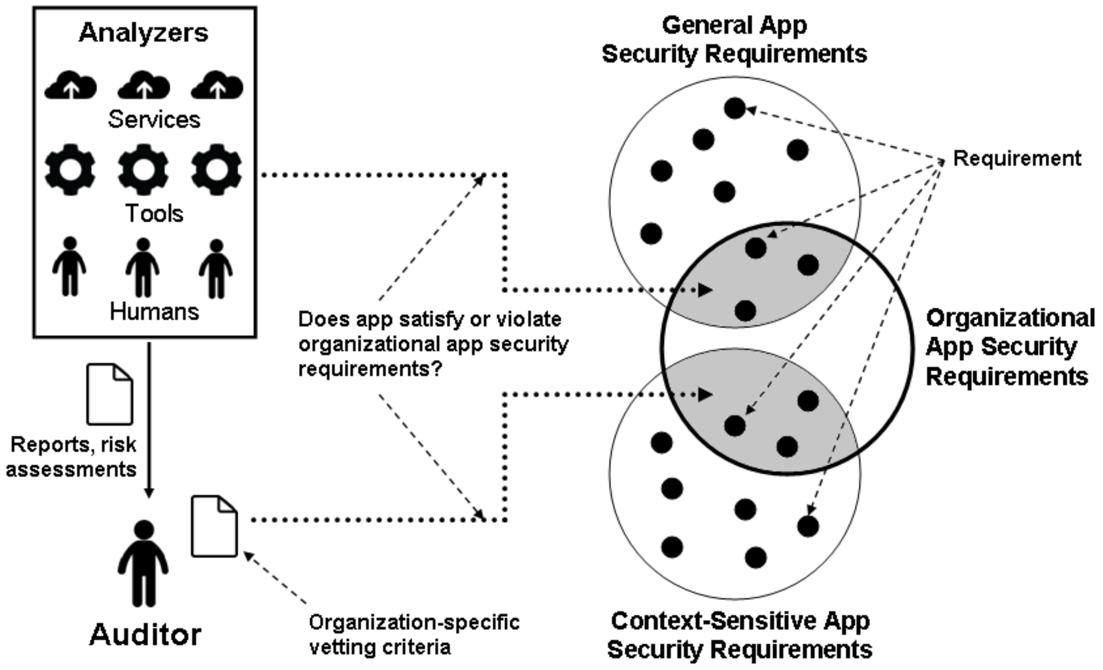
Analyzers
Services
Tools
Humans
Reports, risk assessments
Auditor
Organization-specific vetting criteria
Does app satisfy or violate organizational app security requirements?
General App Security Requirements
Requirement
Organizational App Security Requirements
Context-Sensitive App Security Requirements

Figure 2. Organizational app security requirements.

- Does the organization have specific security and privacy needs, secure behavior expectations, and/or risk management needs? For example, what assets in the organization must be protected, and what events must be avoided? What is the impact if the assets are compromised or the undesired events occur?
- What is the set of users that are permitted to use an app?
- Under what circumstances should an app not be used?
- What vetting has already been performed by the official app store, if known?
- Are the critical assets located on mobile devices, or is the concern that mobile devices will be used as a springboard to attack those assets?
- Are threat assessments being performed?
- What characteristics of attacks and attackers is the organization concerned about? For example:
  - What information about personnel would harm the organization as a whole if it were disclosed?
  - Is there a danger of espionage?
  - Is there a danger of malware designed to disrupt operations at critical moments?
  - What kinds of entities might profit from attacking the organization?

- What is the mobile computing environment? Do wireless devices carried by personnel connect to public providers, a communication infrastructure owned by the organization, or both at different times? How secure is the organization's wireless infrastructure if it has one?
- Is the organization interested only in vetting apps, or is an app vetting process expected to evaluate other aspects of mobile device security such as the operating system (OS), firmware, hardware, and communications? Is the app vetting process meant only to perform evaluations or does it play a broader role in the organization's approach to mobile security? Is the vetting process part of a larger security infrastructure, and if so, what are the other components? Note that only software-related vetting is in the scope of this document, but if the organization expects more than this, the vetting process designers should be aware of it.
- How many apps per week will the app vetting process be expected to handle, and how much variation is permissible in the time needed to process individual apps? High volume combined with low funding might necessitate compromises in the quality of the evaluation. Some organizations may believe that a fully automated assessment pipeline provides more risk reduction than it actually does and therefore expect unrealistically low day-to-day expenses.

Organizations should perform a risk analysis [10] to understand and document the potential security risks in order to help identify appropriate security requirements. Some types of vulnerabilities or weaknesses discovered in apps may be mitigated by other security controls included in the enterprise mobile device architecture. With respect to these other controls, organizations should review and document the mobile device hardware and operating system security controls (e.g., encrypted file systems), to identify which security and privacy requirements can be addressed by the mobile device itself. In addition, mobile enterprise security technologies such as Mobile Device Management (MDM) solutions should also be reviewed to identify which security requirements can be addressed by these technologies.

2.3.2 Understanding Vetting Limitations

As with any software assurance process, there is no guarantee that even the most thorough vetting process will uncover all potential vulnerabilities. Organizations should be made aware that although app security assessments should generally improve the security posture of the organization, the degree to which they do so may not be easily or immediately ascertained. Organizations should also be made aware of what the vetting process does and does not provide in terms of security.

Organizations should also be educated on the value of humans in security assessment processes and ensure that their app vetting does not rely solely on automated tests. Security analysis is primarily a human-driven process [11, 12]; automated tools by themselves cannot address many of the contextual and nuanced interdependencies that underlie software security. The most obvious reason for this is that fully understanding software behavior is one of the classic impossible problems of computer science [13], and in fact current technology has not even reached the limits of what is theoretically possible. Complex, multifaceted software architectures cannot be fully analyzed by automated means.

A further problem is that current software analysis tools do not inherently understand what software has to do to behave in a secure manner in a particular context. For example, failure to encrypt data transmitted to the cloud may not be a security issue if the transmission is tunneled through a virtual private network (VPN). Even if the security requirements for an app have been correctly predicted and are completely understood, there is no current technology for unambiguously translating human-readable requirements into a form that can be understood by machines.

For these reasons, security analysis requires humans (e.g., auditors) in the loop, and by extension the quality of the outcome depends, among other things, on the level of human effort and level of expertise available for an evaluation. Auditors should be familiar with standard processes and best practices for software security assessment (e.g., see [11, 14, 15, 16]). A robust app vetting process should use multiple assessment tools and processes, as well as human interaction, to be successful; reliance on only a single tool, even with human interaction, is a significant risk because of the inherent limitations of each tool.

2.3.3 Budget and Staffing

App software assurance activity costs should be included in project budgets and should not be an afterthought. Such costs may include licensing costs for analyzers and salaries for auditors, approvers, and administrators. Organizations that hire contractors to develop apps should specify that app assessment costs be included as part of the app development process. Note, however, that for apps developed in-house, attempting to implement app assessments solely at the end of the development effort will lead to increased costs and lengthened project timelines. It is strongly recommended to identify potential vulnerabilities or weaknesses during the development process when they can still be addressed by the original developers.

To provide an optimal app vetting process implementation, it is critical for the organization to hire personnel with appropriate expertise. For example, organizations should hire auditors experienced in software security and information assurance as well as administrators experienced in mobile security.

3 App Testing

In the app testing activity, an app is sent by an administrator to one or more analyzers. After receiving an app, an analyzer assumes control of the app in order to preprocess it prior to testing. This initial part of the app testing activity introduces a number of security-related issues, but such issues pertain more to the security of the app file itself rather than the vulnerabilities that might exist in the app. Regardless, such issues should be considered by the organization for ensuring the integrity of the app, protecting intellectual property, and ensuring compliance with licensing and other agreements. Most of these issues are related to unauthorized access to the app.

If an app is submitted electronically by an administrator to an analyzer over a network, issues may arise if the transmission is made over unencrypted channels, potentially allowing unauthorized access to the app by an attacker. After an app is received, an analyzer will often store an app on a local file system, database, or repository. If the app is stored on an unsecured machine, unauthorized users could access the app, leading to potential licensing violations (e.g., if an unauthorized user copies a licensed app for their personal use) and integrity issues (e.g., if the app is modified or replaced with another app by an attacker). In addition, violations of intellectual property may also arise if unauthorized users access the source code of the app. Even if source code for an app is not provided by an administrator, an analyzer will often be required to decompile the (binary) app prior to testing. In both cases, the source or decompiled code will often be stored on a machine by the analyzer. If that machine is unsecured, the app's code may be accessible to unauthorized users, potentially allowing a violation of intellectual property. Note that organizations should review an app’s End User License Agreement (EULA) regarding the sending of an app to a third-party for analysis as well as the reverse engineering of the app’s code by the organization or a third-party to ensure that the organization is not in violation of the EULA.

If an analyzer is external to the organization, then the responsibility of ensuring the security of the app file falls on the analyzer. Thus, the best that an organization can do to ensure the integrity of apps, the protection of intellectual property, and the compliance with licensing agreements is to understand the security implications surrounding the transmission, storage, and processing of an app by the specific analyzer.

After an app has been preprocessed by an analyzer, the analyzer will then test the app for software vulnerabilities that violate one or more general app security requirements. In this section, we describe these general requirements and the testing approaches used to detect vulnerabilities that violate these requirements. Descriptions of specific Android and iOS app vulnerabilities are provided in Appendices B and C. In addition, recommendations related to the app testing activity are described in Appendix A.2. For more information on security testing and assessment tools and techniques that may be helpful for performing app testing, see [17].

3.1 General Requirements

A general requirement is an app security requirement that specifies a software characteristic or behavior that an app should exhibit in order to be considered secure. General app security requirements include:

- **Enabling authorized functionality:** The app must work as described; all buttons, menu items, and other interfaces must work. Error conditions must be handled gracefully, such as when a service or function is unavailable (e.g., disabled, unreachable, *etc.*).
- **Preventing unauthorized functionality:** Unauthorized functionality, such as data exfiltration performed by malware, must not be supported.

- **Limiting permissions:** Apps should have only the minimum permissions necessary and should only grant other applications the necessary permissions.
- **Protecting sensitive data:** Apps that collect, store, and transmit sensitive data should protect the confidentiality and integrity of this data. This category includes preserving privacy, such as asking permission to use personal information and using it only for authorized purposes.
- **Securing app code dependencies:** The app must use any dependencies, such as on libraries, in a reasonable manner and not for malicious reasons.
- **Testing app updates:** New versions of the app must be tested to identify any new weaknesses.

Well-written security requirements are most useful to auditors when they can easily be translated into specific evaluation activities. The various processes concerning the elicitation, management, and tracking of requirements are collectively known as requirements engineering [18, 19], and this is a relatively mature activity with tools support. Presently, there is no methodology for driving all requirements down to the level where they could be checked completely[3] by automatic software analysis, while ensuring that the full scope of the original requirements is preserved. Thus, the best we can do may be to document the process in such a way that human reviews can find gaps and concentrate on the types of vulnerabilities, or weaknesses, that fall into those gaps.

3.1.1 Enabling Authorized Functionality

An important part of confirming authorized functionality is testing the mobile user interface (UI) display, which can vary greatly for different device screen sizes and resolutions. The rendering of images or position of buttons may not be correct due to these differences. If applicable, the UI should be viewed in both portrait and landscape mode. If the page allows for data entry, the virtual keyboard should be invoked to confirm it displays and works as expected. Analyzers should also test any device physical sensors used by the app such as Global Positioning System (GPS), front/back cameras, video, microphone for voice recognition, accelerometers (or gyroscopes) for motion and orientation-sensing, and communication between devices (e.g., phone “bumping”).

Telephony functionality encompass a wide variety of method calls that an app can use if given the proper permissions, including making phone calls, transmitting Short Message Service (SMS) messages, and retrieving unique phone identifier information. Most, if not all of these telephony events are sensitive, and some of them provide access to PII. Many apps make use of the unique phone identifier information in order to keep track of users instead of using a username/password scheme. Another set of sensitive calls can give an app access to the phone number. Many legitimate apps use the phone number of the device as a unique identifier, but this is a bad coding practice and can lead to loss of PII. To make matters worse, many of the carrier companies do not take any caution in protecting this information. Any app that makes use of the phone call or SMS messages that is not clearly stated in the EULA or app description as intending to use them should be immediate cause for suspicion.

3.1.2 Preventing Unauthorized Functionality

Some apps (as well as their libraries) are malicious and intentionally perform functionality that is not disclosed to the user and violates most expectations of secure behavior. This undocumented functionality may include exfiltrating confidential information or PII to a third party, defrauding the user by sending premium SMS messages (premium SMS messages are meant to be used to pay for products or services),

[3] In the mathematical sense of “completeness,” completeness means that no violations of the requirement will be overlooked.

or tracking users' locations without their knowledge. Other forms of malicious functionality include injection of fake websites into the victim's browser in order to collect sensitive information, acting as a starting point for attacks on other devices, and generally disrupting or denying operation. Another example is the use of banner ads that may be presented in a manner which causes the user to unintentionally select ads that may attempt to deceive the user. These types of behaviors support phishing attacks and may not be detected by mobile antivirus or software vulnerability scanners as they are served dynamically and not available for inspection prior to the installation of the app.

Open source and commercially available malware detection and analysis tools can identify both known and new forms of malware. These tools can be incorporated as part of an organization's enterprise mobile device management (MDM) solution, organization's app store, or app vetting process. Note that even if these tools have not detected known malware in an app, one should not assume that malicious functionality is not present. It is essentially impossible for a vetting process to guarantee that a piece of software is free from malicious functionality. Mobile devices may have some built-in protections against malware, for example app sandboxing and user approval in order to access subsystems such as the GPS or text messaging. However, sandboxing only makes it more difficult for apps to interfere with one another or with the operating system; it does not prevent many types of malicious functionality.

A final category of unauthorized functionality to consider is communication with disreputable websites, domains, servers, *etc.* If the app is observed to communicate with sites known to harbor malicious advertising, spam, phishing attacks, or other malware, this is a strong indication of unauthorized functionality.

3.1.3 Limiting Permissions

Some apps have permissions that are not consistent with EULAs, app permissions, app descriptions, in-program notifications, or other expected behaviors and would not be considered to exhibit secure behavior. An example is a wallpaper app that collects and stores sensitive information, such as passwords or PII, or accesses the camera and microphone. Although these apps might not have malicious intent—they may just be poorly designed—their excessive permissions still expose the user to threats that stem from the mismanagement of sensitive data and the loss or theft of a device. Similarly, some apps have permissions assigned that they don't actually use. Moreover, users routinely reflexively grant whatever access permissions a newly installed app requests, should their mobile device OS perform permission requests. It is important to note that identifying apps that have excessive permissions is a manual process. It involves a subjective decision from an auditor that certain permissions are not appropriate and that the behavior of the app is insecure.

Excessive permissions can manifest in other ways including:

- **File input/output (I/O) and removable storage:** File I/O can be a security risk, especially when the I/O happens on a removable or unencrypted portion of the file system. Regarding the app's own behavior, file scanning or access to files that are not part of an app's own directory could be an indicator of malicious activity or bad coding practice. Files written to external storage, such as a removable Secure Digital (SD) card, may be readable and writeable by other apps that may have been granted different permissions, thus placing data written to unprotected storage at risk.

- **Privileged commands:** Apps may possess the ability to invoke lower-level command line programs, which may allow access to low-level structures, such as the root directory, or may allow access to sensitive commands. These programs potentially allow a malicious app access to various system resources and information (e.g., finding out the running processes on a device). Although the mobile operating system typically offers protection against directly accessing resources beyond what is available to the user account that the app is running under, this opens up the potential for privilege elevation attacks.

- **APIs:** The app only uses designated APIs from the vendor-provided software development kit (SDK) and uses them properly; no other API calls are permitted.[4] For the permitted APIs, the analyzer should note where the APIs are transferring data to and from the app.

3.1.4 Protecting Sensitive Data

Many apps collect, store, and/or transmit sensitive data, such as financial data (e.g., credit card numbers), personal data (e.g., social security numbers), and login credentials (e.g., passwords). When implemented and used properly, cryptography can help maintain the confidentiality and integrity of this data, even if a protected mobile device is lost or stolen and falls into an attacker's hands. However, only certain cryptographic implementations have been shown to be sufficiently strong, and developers who create their own implementations typically expose their apps to exploitation through man-in-the-middle attacks and other forms.

Federal Information Processing Standard (FIPS) 140-2 precludes the use of unvalidated cryptography for the cryptographic protection of sensitive data within federal systems. Unvalidated cryptography is viewed by NIST as providing no protection to the data. A list of FIPS 140-2-validated cryptographic modules and an explanation of the applicability of the Cryptographic Module Validation Program (CMVP) can be found on the CMVP website [20]. Analyzers should reference the CMVP-provided information to determine if each app is using properly validated cryptographic modules and approved implementations of those modules.

It is not sufficient to use appropriate cryptographic algorithms and modules; cryptography implementations must also be maintained in accordance with approved practices. Cryptographic key management is often performed improperly, leading to exploitable weaknesses. For example, the presence of hard-coded cryptographic keying material such as keys, initialization vectors, *etc.* is an indication that cryptography has not been properly implemented by an app and might be exploited by an attacker to compromise data or network resources. Guidelines for proper key management techniques can be found in [21]. Another example of a common cryptographic implementation problem is the failure to properly validate digital certificates, leaving communications protected by these certificates subject to man-in-the-middle attacks.

Privacy considerations need to be taken into account for apps that handle PII, including mobile-specific personal information like location data and pictures taken by onboard cameras, as well as the broadcast ID of the device. This needs to be dealt with in the UI as well as in the portions of the apps that manipulate this data. For example, an analyzer can verify that the app complies with privacy standards by masking characters of any sensitive data within the page display, but audit logs should also be reviewed when possible for appropriate handling of this type of information. Another important privacy consideration is that sensitive data should not be disclosed without prior notification to the user by a prompt or a license agreement.

Another concern with protecting sensitive data is data leakage. For example, data is often leaked through unauthorized network connections. Apps can use network connections for legitimate reasons, such as

[4] The existence of an API raises the possibility of malicious use. Even if the APIs are used properly, they may pose risks because of covert channels, unintended access to other APIs, access to data exceeding original design, and the execution of actions outside of the app's normal operating parameters.

retrieving content, including configuration files, from a variety of external systems. Apps can receive and process information from external sources and also send information out, potentially exfiltrating data from the mobile device without the user's knowledge. Analyzers should consider not only cellular and Wi-Fi usage, but also Bluetooth, NFC, and other forms of networking. Another common source of data leakage is shared system-level logs, where multiple apps log their security events to a single log file. Some of these log entries may contain sensitive information, either recorded inadvertently or maliciously, so that other apps can retrieve it. Analyzers should examine app logs to look for signs of sensitive data leakage.

3.1.5 Securing App Code Dependencies

In general, an app must not use any unsafe coding or building practices. Specifically, an app should properly use other bodies of code, such as libraries, only when needed, and not to attempt to obfuscate malicious activity.[5] Examples include:

- **Native Methods:** Native method calls are typically calls to a library function that has already been loaded into memory. These methods provide a way for an app to reuse code that was written in a different language. These calls, however, can provide a level of obfuscation that impacts the ability to perform analysis of the app.
- **External Libraries and Classes:** This category includes any third-party libraries and classes that are loaded by the app at run time. Third-party libraries and classes can be closed source, can have self-modifying code, or can execute unknown server-side code. Legitimate uses for loaded libraries and classes might be for the use of a cryptographic library or a graphics API. Malicious apps can use library and class loading as a method to avoid detection. From a vetting perspective, libraries and classes are also a concern because they introduce outside code to an app without the direct control of the developer. Libraries and classes can have their own known vulnerabilities, so an app using such libraries or classes could expose a known vulnerability on an external interface. Tools are available that can list libraries, and these libraries can then be searched for in vulnerability databases to determine if they have known weaknesses.
- **Dynamic Behavior:** When apps execute, they exhibit a variety of dynamic behaviors. Not all of these operating behaviors are a result of user input. Executing apps may also receive inputs from data stored on the device. The key point here is the need to know where data used by an app originates from and knowing whether and how it gets sanitized. It is critical to recognize that data downloaded from an external source is particularly dangerous as a potential exploit vector unless it is clear how the app prevents insecure behaviors resulting from data from a source not trusted by the organization using the app. Data downloaded from external sources should be treated as potential exploits unless it can be shown that the app is able to thwart any negative consequences from externally supplied data, especially data from untrusted sources. This is nearly impossible, thus requiring some level of risk-tolerance mitigation.
- **Inter-Application Communications:** Apps that communicate with each other can provide useful capabilities and productivity improvements, but these inter-application communications can also present a security risk. For example, in Android platforms, inter-application communications are allowed but regulated by what Android calls "intents." An intent can be used to start an app component in a different app or the same app that is sending the intent. Intents can also be used to

[5] There are both benign and malicious reasons for obfuscating code (e.g., protecting intellectual property and concealing malware, respectively). Code obfuscation makes static analysis difficult, but dynamic analysis can still be effective.

interact and request resources from the operating system. In general, analyzers should be aware of situations where an app is using nonhuman entities to make API calls to other devices, to communicate with third-party services, or to otherwise interact with other systems.

3.1.6 Testing App Updates

To provide long-term assurance of the software throughout its life cycle, all apps, as well as their updates, should go through a software assurance vetting process, because each new version of an app can introduce new unintentional weaknesses or unreliable code. Apps should be vetted before being released to the community of users within the organization. The purpose is to validate that the app adheres to a predefined acceptable level of security risk and identify whether the developers have introduced any latent weaknesses that could make the IT infrastructure vulnerable.

It is increasingly common for mobile devices to be capable of and be configured to automatically download app updates. Mobile devices are also generally capable of downloading apps of the user's choice from app stores. Ideally, each app and app update should be vetted before it is downloaded onto any of the organization's mobile devices. If unrestricted app and app update downloads are permitted, this can bypass the vetting process. There are several options to work around this, depending on the mobile device platform and the degree to which the mobile devices are managed by the organization. Possible options include disabling all automatic updates, only permitting use of an organization-provided app store, preventing the installation of updates through the use of application whitelisting software, and enabling MDM mobile application management features that make app vetting a precondition for allowing users to download apps and app updates. Further discussion of this is out of scope because it is largely addressed operationally as part of OS security practices.

3.2 Testing Approaches

To detect the satisfaction or violation of a general requirement, an analyzer tests an app for the presence of software vulnerabilities. Such testing may involve (1) correctness testing, (2) analysis of the app's source code or binary code, (3) the use of static or dynamic analysis, and (4) manual or automatic testing of the app.

3.2.1 Correctness Testing

One approach for testing an app is software correctness testing [22]. Software correctness testing is the process of executing a program with the intent of finding errors. Although software correctness testing is aimed primarily at improving quality assurance, verifying and validating described functionality, or estimating reliability, it can also help to reveal potential security vulnerabilities since such vulnerabilities often have a negative effect on the quality, functionality, and reliability of the software. For example, software that crashes or exhibits unexpected behavior is often indicative of a security flaw. One of the advantages of software correctness testing is that it is traditionally based on specifications of the specific software to be tested. These specifications can then be transformed into requirements that specify how the software is expected to behave under test. This is distinguished from security assessment approaches that often require the tester to derive requirements themselves; often such requirements are largely based on security requirements that are considered to be common across many different software artifacts and may not test for vulnerabilities that are unique to the software under test. Nonetheless, because of the tight coupling between security and the quality, functionality, and reliability of software, it is recommended that software correctness testing be performed when possible.

3.2.2 Source Code Versus Binary Code

A major factor in performing app testing is whether source code is available. Typically, apps downloaded from an app store do not provide access to source code. When source code is available, such as in the case of an open source app, a variety of tools can be used to analyze it. The goals of performing a source code review are to find vulnerabilities in the source code and to verify the results of analyzers. The current practice is that these tasks are performed manually by a secure code reviewer who reads through the contents of source code files. Even with automated aids, the analysis is labor-intensive. Benefits to using automated static analysis tools include introducing consistency between different reviews and making review of large codebases possible. Reviewers should generally use automated static analysis tools whether they are conducting an automated or a manual review, and they should express their findings in terms of Common Weakness Enumeration (CWE) identifiers or some other widely accepted nomenclature. Performing a secure code review requires software development and domain-specific knowledge in the area of application security. Organizations should ensure that the individuals performing source code reviews have the necessary skills and expertise. Note that organizations that intend to develop apps in-house should also refer to guidance on secure programming techniques and software quality assurance processes to appropriately address the entire software development life cycle [11, 23].

When source code is not available, binary code can be vetted instead. In the context of apps, the term “binary code” can refer to either byte-code or machine code. For example, Android apps are compiled to byte-code that is executed on a virtual machine, similar to the Java Virtual Machine (JVM), but they can also come with custom libraries that are provided in the form of machine code, that is, code executed directly on the mobile device's CPU. Android binary apps include byte-code that can be analyzed without hardware support using emulated and virtual environments.

3.2.3 Static Versus Dynamic Analysis

Analysis tools are often characterized as being either *static* or *dynamic*.[6] Static analysis examines the app source code and binary code, and attempts to reason over all possible behaviors that might arise at runtime. It provides a level of assurance that analysis results are an accurate description of the program’s behavior regardless of the input or execution environment. Dynamic analysis operates by executing a program using a set of input use cases and analyzing the program’s runtime behavior. In some cases, the enumeration of input test cases is large, resulting in lengthy processing times. However, methods such as combinatorial testing can reduce the number of dynamic input test case combinations, reducing the amount of time needed to derive analysis results [25]. Still, dynamic analysis is unlikely to provide 100 percent code coverage [26]. Organizations should consider the technical trade-off differences between what static and dynamic tools offer and balance their usage given the organization’s software assurance goals.

Static analysis requires that binary code be reverse engineered when source is not available, which is relatively easy for byte-code,[7] but can be difficult for machine code. Many commercial static analysis tools already support byte-code, as do a number of open source and academic tools.[8] For machine code, it is especially hard to track the flow of control across many functions and to track data flow through

[6] For mobile devices, there are analysis tools that label themselves as performing behavioral testing. Behavioral testing (also known as behavioral analysis) is a form of static and dynamic testing that attempts to detect malicious or risky behavior, such as the oft-cited example of a flashlight app that accesses a contact list. [24] This publication assumes that any mention of static or dynamic testing also includes behavioral testing as a subset of its capabilities.

[7] The ASM framework [27] is a commonly used framework for byte-code analysis.

[8] Such as [28, 29, 30, 31].

variables, since most variables are stored in anonymous memory locations that can be accessed in different ways. The most common way to reverse engineer machine code is to use a disassembler or a decompiler that tries to recover the original source code. These techniques are especially useful if the purpose of reverse engineering is to allow humans to examine the code, since the outputs are in a form that can be understood by humans with appropriate skills. But even the best disassemblers make mistakes [32], and some of those mistakes can be corrected with formal static analysis. If the code is being reverse engineered for static analysis, then it is often preferable to disassemble the machine code directly to a form that the static analyzer understands rather than creating human-readable code as an intermediate byproduct. A static analysis tool that is aimed at machine code is likely to automate this process.

For dynamic testing (as opposed to static analysis), the most important requirement is to be able to see the workings of the code as it is being executed. There are two primary ways to obtain this information. First, an executing app can be connected to a remote debugger, and second, the code can be run on an emulator that has debugging capabilities built into it. Running the code on a physical device allows the analyzer to select the exact characteristics of the device on which the app is intended to be used and can provide a more accurate view about how the app will be behave. On the other hand, an emulator provides more control, especially when the emulator is open source and can be modified by the evaluator to capture whatever information is needed. Although emulators can simulate different devices, they do not simulate all of them, and the simulation may not be completely accurate. Note that malware increasingly detects the use of emulators as a testing platform and changes its behavior accordingly to avoid detection. Therefore, it is recommended that analyzers use a combination of emulated and physical mobile devices so as to avoid false negatives from malware that employs anti-detection techniques.

Useful information can be gleaned by observing an app’s behavior even without knowing the purposes of individual functions. For example, an analyzer can observe how the app interacts with its external resources, recording the services it requests from the operating system and the permissions it exercises. Note that although many of the device capabilities used by an app may be inferred by an analyzer (e.g., access to a device's camera will be required of a camera app), an app may be permitted access to additional device capabilities that are beyond the scope of its described functionality (e.g., a camera app accessing the device's network). Moreover, if the behavior of the app is observed for specific inputs, the evaluator can ask whether the capabilities being exercised make sense in the context of those particular inputs. For example, a calendar app may legitimately have permission to send calendar data across the network in order to sync across multiple devices, but if the user has merely asked for a list of the day’s appointments and the app sends data that is not simply part of the handshaking process needed to retrieve data, the analyzer might investigate what data is being sent and for what purpose.

3.2.4 Automated Tools

In most cases, analyzers used by an app vetting process will consist of automated tools that test an app for software vulnerabilities and generate a report and risk assessment. Classes of automated tools include:

- **Simulators:** Desktop simulators allow the use of a computer to view how the app will display on a specific device without the use of an actual device. Because the tool provides access to the UI, the app features can also be tested. However, interaction with the actual device hardware features such as a camera or accelerometer cannot be simulated and requires an actual device.
- **Remote Device Access:** These types of tools allow the analyzer to view and access an actual device from a computer. This allows the testing of most device features that do not require physical movement such as using a video camera or making a phone call. Remote debuggers are a common way of examining and understanding apps.

- **Automated Testing:** Basic mobile functionality testing lends itself well to automated testing. There are several tools available that allow creation of automated scripts to remotely run regression test cases on specific devices and operating systems.
    - **User Interface-Driven Testing:** If the expected results being verified are UI-specific, pixel verification can be used. This method takes a "screen shot" of a specified number of pixels from a page of the app and verifies that it displays as expected (pixel by pixel).
    - **Data-Driven Testing:** Data-driven verification uses labels or text to identify the section of the app page to verify. For example, it will verify the presence of a "Back" button, regardless of where on the page it displays. Data-driven automated test cases are less brittle and allow for some page design change without rewriting the script.
    - **Fuzzing:** Fuzzing normally refers to the automated generation of test inputs, either randomly or based on configuration information describing data format. Fault injection tools may also be referred to as "fuzzers." This category of test tools is not necessarily orthogonal to the others, but its emphasis is on fast and automatic generation of many test scenarios.
    - **Network-Level Testing:** Fuzzers, penetration test tools, and human-driven network simulations can help determine how the app interacts with the outside world. It may be useful to run the app in a simulated environment during network-level testing so that its response to network events can be observed more closely.
- **Test Automation:** The term "test automation" usually refers to tools that automate the repetition of tests after they have been engineered by humans. This makes it useful for tests that need to be repeated often (e.g., tests that will be used for many apps or executed many times, with variations, for a single app).
- **Static Tools:**
    - **Static Analysis Tools:** These tools generally analyze the behavior of software (either source or binary code) to find software vulnerabilities, though—if appropriate specifications are available—static analysis can also be used to evaluate correctness. Some static analysis tools operate by looking for syntax embodying a known class of potential vulnerabilities and then analyzing the behavior of the program to determine whether the weaknesses can be exploited. Static tools should encompass the app itself, but also the data it uses to the extent that this is possible; keep in mind that the true behavior of the app may depend critically on external resources. [9]
    - **Metrics Tools:** These tools measure aspects of software not directly related to software behavior but useful in estimating auxiliary information such as the effort of code evaluation. Metrics can also give indirect indications about the quality of the design and development process.

Commercial automated app testing tools have overlapping or complementary capabilities. For example, one tool may be based on techniques that find integer overflows [34] reliably while another tool may be better at finding weaknesses related to command injection attacks [35]. Finding the right set of tools to evaluate a requirement can be a challenging task because of the varied capabilities of diverse commercial

[9] The Software Assurance Metrics and Tool Evaluation (SAMATE) test suite [33] provides a baseline for evaluating static analysis tools. Tool vendors may evaluate their own tools with SAMATE, so it is to be expected that, over time, the tools will eventually perform well on precisely the SAMATE specified tests. Still, the SAMATE test suite can help determine if the tool is meant to do what analyzers thought, and the SAMATE test suite is continually evolving

and open source tools, and because it can be challenging to determine the capabilities of a given tool. Constraints on time and money may also prevent the evaluator from assembling the best possible evaluation process, especially when the best process would involve extensive human analysis. Tools that provide a high-level risk rating should provide transparency on how the score was derived and which tests were performed. Tools that provide low-level code analysis reports should help analyzers understand how the tool's findings may impact their security. Therefore, it is important to understand and quantify what each tool and process can do, and to use that knowledge to characterize what the complete evaluation process does or does not provide.

In most cases, organizations will need to use multiple automated tools to meet their testing requirements. Using multiple tools can provide improved coverage of software vulnerability detection as well as provide validation of test results for those tools that provide overlapping tests (i.e., if two or more tools test for the same vulnerability).

3.3 Sharing Results

The sharing of an organization's findings for an app can greatly reduce the duplication and cost of app vetting efforts for other organizations. Information sharing within the software assurance community is vital and can help analyzers benefit from the collective efforts of security professionals around the world. The National Vulnerability Database (NVD) [36] is the U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP) [37]. This data enables automation of vulnerability management, security measurement, and compliance. The NVD includes databases of security checklists, security-related software flaws, misconfigurations, product names, and impact metrics. SCAP is a suite of specifications that standardize the format and nomenclature by which security software products communicate software flaw and security configuration information. SCAP is a multipurpose protocol that supports automated vulnerability checking, technical control compliance activities, and security measurement. Goals for the development of SCAP include standardizing system security management, promoting interoperability of security products, and fostering the use of standard expressions of security content. The CWE [38] and Common Attack Pattern Enumeration and Classification (CAPEC) [39] collections can provide a useful list of weaknesses and attack approaches to drive a binary or live system penetration test. Classifying and expressing software vulnerabilities is an ongoing and developing effort in the software assurance community, as is how to prioritize among the various weaknesses that can be in an app [40] so that an organization can know that those that pose the most danger to the app, given its intended use/mission, are addressed by the vetting activity given the difference in the effectiveness and coverage of the various available tools and techniques.

4 App Approval/Rejection

After an app has been tested by an analyzer, the analyzer generates a report that identifies detected software vulnerabilities and a risk assessment that expresses the estimated level of risk associated with using the app. The report and risk assessment are then made available to one or more auditors of the organization. In this section, we describe issues surrounding the auditing of report and risk assessments by an auditor, the organization-specific vetting criteria used by auditors to determine if an app meets the organization's context-sensitive requirements, and issues surrounding the final decision by an approver to approve or reject an app. Recommendations related to the app approval/rejection activity are described in Appendix A.3

4.1 Report and Risk Auditing

An auditor inspects reports and risk assessments from one or more analyzers to ensure that the app meets the organization's general security requirements. To accomplish this, the auditor must be intimately familiar with the organization's general security requirements as well as with the reports and risk assessments from analyzers. After inspecting analyzers' reports and risk assessments against general app security requirements, the auditor also assesses the app with respect to context-sensitive requirements using organization-specific vetting criteria. After assessing the app's satisfaction or violation of both general and context-specific security requirements, the auditor then generates a recommendation for approving or rejecting the app for deployment based on the app's security posture and makes this available to the organization's approver.

One of the main issues related to report and risk auditing stems from the difficulty in collating and interpreting different reports and risk assessments from multiple analyzers due to the wide variety of security-related definitions, semantics, nomenclature, and metrics. For example, an analyzer may classify the estimated risk for using an app as *low*, *moderate*, *high*, or *severe* risk while another analyzer classifies estimated risk as *pass*, *warning*, or *fail*. Note that it is in the best interest of analyzers to provide assessments that are relatively intuitive and easy to interpret by auditors. Otherwise, organizations will likely select other analyzers that generate reports and assessments that their auditors can understand. While some standards exist for expressing risk assessment (e.g., the Common Vulnerability Scoring System [41]) and vulnerability reports (e.g., Common Vulnerability Reporting Framework [42]), the current adoption of these standards by analyzers is low. To address this issue, it is recommended that an organization identify analyzers that leverage vulnerability reporting and risk assessment standards. If this is not possible, it is recommended that the organization provide sufficient training to auditors on both the security requirements of the organization as well as the interpretation of reports and risk assessments generated by analyzers.

The methods used by an auditor to derive a recommendation for approving or rejecting an app is based on a number of factors including the auditor's confidence in the analyzers' assessments and the perceived level of risk from one or more risk assessments as well as the auditor's understanding of the organization's risk threshold, the organization's general and context-sensitive security requirements, and the reports and risk assessments from analyzers. In some cases, an organization will not have any context-sensitive requirements and thus, auditors will evaluate the security posture of the app based solely on reports and risk assessments from analyzers. In such cases, if risk assessments from all analyzers indicate a low security risk associated with using an app, an auditor may adequately justify the approval of the app based on those risk assessments (assuming that the auditor has confidence in the analyzers' assessments). Conversely, if one or more analyzers indicate a high security risk with using an app, an auditor may adequately justify the rejection of the app based on those risk assessments. Cases where moderate, but no high, levels of risk are assessed for an app by one or more analyzers will require additional evaluation by the auditor in order to adequately justify a recommendation for approving or rejecting the app. To increase the likelihood of an appropriate recommendation, it is recommended that the organization use multiple auditors.

4.2 Organization-Specific Vetting Criteria

As described in Section 2.3.1, a context-sensitive requirement is an app security requirement that specifies how apps should be used by the organization to ensure that organization's security posture. For an app, the satisfaction or violation of context-sensitive requirements cannot be determined by analyzers but instead must be determined by an auditor using organization-specific vetting criteria. Such criteria include:

- **Requirements:** The pertinent requirements, security policies, privacy policies, acceptable use policies, and social media guidelines that are applicable to the organization.
- **Provenance:** Identity of the developer, developer’s organization, developer’s reputation, date received, marketplace/app store consumer reviews, *etc.*
- **Data Sensitivity:** The relative sensitivity of the data collected, stored, and/or transmitted by the app.
- **App Criticality:** How critical the app is to the organization’s business processes.
- **Target Users:** The intended set of users of the app.
- **Target Hardware**: The intended hardware platform and configuration on which the app will be deployed.
- **Target Environment**: The intended operational environment of the app (e.g., general public use vs. sensitive military environment).
- **Digital Signature**: Digital signatures applied to the app binaries or packages.[10]
- **App Documentation:**
  - **User Guide:** When available, the app’s user guide assists testing by specifying the expected functionality and expected behaviors. This is simply a statement from the developer describing what they claim their app does and how it does it.
  - **Test Plans:** Reviewing the developer’s test plans may help focus app vetting by identifying any areas that have not been tested or were tested inadequately. A developer could opt to submit a test oracle in certain situations to demonstrate its internal test effort.
  - **Testing Results:** Code review results and other testing results will indicate which security standards were followed. For example, if an application threat model was created, this should be submitted. This will list weaknesses that were identified and should have been addressed during design and coding of the app.

[10] The level of assurance provided by digital signatures varies widely. For example, one organization might have stringent digital signature requirements that provide a high degree of trust, while another organization might allow self-signed certificates to be used, which do not provide any level of trust.

- **Service-Level Agreement:** If an app was developed for an organization by a third party, a Service-Level Agreement (SLA) may have been included as part of the vendor contract. This contract should require the app to be compatible with the organization's security policy.

Some information can be gleaned from app documentation in some cases, but even if documentation does exist, it might lack technical clarity and/or use jargon specific to the circle of users who would normally purchase the app. Since the documentation for different apps will be structured in different ways, it may also be time-consuming to find the information for evaluation. Therefore, a standardized questionnaire might be appropriate for determining the software's purpose and assessing an app developer's efforts to address security weaknesses. Such questionnaires aim to identify software quality issues and security weaknesses by helping developers address questions from end users/adopters about their software development processes. For example, developers can use the Department of Homeland Security (DHS) Custom Software Questionnaire [43] to answer questions such as *Does your software validate inputs from untrusted resources?* and *What threat assumptions were made when designing protections for your software?* Another useful question (not included in the DHS questionnaire) is *Does your app access a network application programming interface (API)?* Note that such questionnaires are feasible to use only in certain circumstances (e.g., when source code is available and when developers can answer questions).

Known flaws in app design and coding may be reported in publicly accessible vulnerability databases, such as NVD.[11] Before conducting the full vetting process for a publicly available app, auditors should check one or more vulnerability databases to determine if there are known flaws in the corresponding version of the app. If one or more serious flaws have already been discovered, this alone might be sufficient grounds to reject the version of the app for organizational use, thus allowing the rest of the vetting process to be skipped. However, in most cases such flaws will not be known, and the full vetting process will be needed. This is so because there are many forms of vulnerabilities other than known flaws in app design and coding. Identifying these weaknesses necessitates first defining the app requirements, so that deviations from these requirements can be flagged as weaknesses.

4.3 Final Approval/Rejection

Ultimately, an organization's decision to approve or reject an app for deployment rests on the decision of the organization's approver. After an approver receives recommendations from one or more auditors that describe the security posture of the app, the approver will consider this information as well as additional non-security-related criteria including budgetary, policy, and mission-specific criteria to render the organization's official decision to approve or reject the app. After this decision is made, the organization should then follow procedures for handling the approval or rejection of the app. These procedures must be specified by the organization prior to implementing an app vetting process. If an app is approved, procedures must be defined that specify how the approved app should be processed and ultimately deployed onto the organization's devices. For example, a procedure may specify the steps needed to digitally sign an app before being submitted to the administrator for deployment onto the organization's devices. Similarly, if an app is rejected, a procedure may specify the steps needed to identify an alternative app or to resolve detected vulnerabilities with the app developer. Procedures that define the steps associated with approving or rejecting an app should be included in the organization's app security policies.

[11] Vulnerability databases generally reference vulnerabilities by their Common Vulnerabilities and Exposures (CVE) identifier. For more information on CVE, see [44].

Appendix A—Recommendations

This appendix describes recommendations for vetting the security of mobile applications. These recommendations are described within the context of the planning of an app vetting process implementation as well as the app testing and app approval/rejection activities.

A.1 Planning

- Perform a risk analysis [10] to understand and document the potential security impact of mobile apps on the organization's computing, networking, and data resources).
- Review and document the mobile device hardware and operating system security controls, for example an encrypted file system, and identify which security and privacy requirements can be addressed by the mobile device itself.
- Review and document mobile enterprise security technologies, such as Mobile Device Management (MDM) solutions, and identify which security and privacy requirements can be addressed by these technologies.
- Review the organization's mobile security architecture and understand what threats are mitigated through the technical and operational controls. Identify potential security and privacy risks that are not mitigated through these technical and operational controls.
- Develop organizational app security requirements by identifying general and context-sensitive requirements.
- Educate organizational staff on the limitations of app vetting and the value of human involvement in an app vetting process.
- Procure an adequate budget for performing an app vetting process.
- Hire personnel, particularly auditors, with appropriate expertise.

A.2 App Testing

- Review licensing agreements associated with analyzers and understand the security implications surrounding the integrity, intellectual property, and licensing issues when submitting an app to an analyzer.
- Ensure that apps transmitted over the network use an encrypted channel (e.g., SSL) and that apps are stored on a secure machine at the analyzer's location. In addition, ensure that only authorized users have access to that machine.
- Identify general app security requirements needed by the organization.
- Select appropriate testing tools and methodologies for determining the satisfaction or violation of general app security requirements.
- Ensure that app updates are tested.
- Leverage existing testing results where possible.

A.3 App Approval/Rejection

- Use analyzers that leverage a standardized reporting format or risk assessment methodology, or that provides intuitive and easy-to-interpret reports and risk assessments.

- Ensure sufficient training of auditors on both the organization's security requirements and interpretation of analyzer reports and risk assessments. Use multiple auditors to increase likelihood of appropriate recommendations.

- Identify organization-specific vetting criteria necessary for vetting context-sensitive app security requirements.

- Monitor public databases, mailing lists, and other publicly available security vulnerability reporting repositories to keep abreast of new developments that may impact the security of mobile devices and mobile apps.

Appendix B—Android App Vulnerability Types

This appendix identifies vulnerabilities specific to apps running on Android mobile devices. The scope of this appendix includes app vulnerabilities for Android-based mobile devices running apps written in Java. The scope does not include vulnerabilities in the mobile platform hardware and communications networks. Although some of the vulnerabilities described below are common across mobile device environments, this appendix focuses only on Android-specific vulnerabilities.

The vulnerabilities in this appendix are broken into three hierarchical levels, A, B, and C. The A level is referred to as the vulnerability class and is the broadest description for the vulnerabilities specified under that level. The B level is referred to as the sub-class and attempts to narrow down the scope of the vulnerability class into a smaller, common group of vulnerabilities. The C level specifies the individual vulnerabilities that have been identified. The purpose of this hierarchy is to guide the reader to finding the type of vulnerability they are looking for as quickly as possible.

The A level general categories of Android app vulnerabilities are listed below:

Table 1. Android Vulnerabilities, A Level

Type	Description	Negative Consequence
Incorrect Permissions	Permissions allow accessing controlled functionality such as the camera or GPS and are requested in the program. Permissions can be implicitly granted to an app without the user's consent.	An app with too many permissions may perform unintended functions outside the scope of the app's intended functionality. Additionally, the permissions are vulnerable to hijacking by another app. If too few permissions are granted, the app will not be able to perform the functions required.
Exposed Communications	Internal communications protocols are the means by which an app passes messages internally within the device, either to itself or to other apps. External communications allow information to leave the device.	Exposed internal communications allow apps to gather unintended information and inject new information. Exposed external communication (data network, Wi-Fi, Bluetooth, NFC, *etc.*) leave information open to disclosure or man-in-the-middle attacks.
Potentially Dangerous Functionality	Controlled functionality that accesses system-critical resources or the user's personal information. This functionality can be invoked through API calls or hard coded into an app.	Unintended functions could be performed outside the scope of the app's functionality.
App Collusion	Two or more apps passing information to each other in order to increase the capabilities of one or both apps beyond their declared scope.	Collusion can allow apps to obtain data that was unintended such as a gaming app obtaining access to the user's contact list.
Obfuscation	Functionality or control flows that are hidden or obscured from the user. For the purposes of this appendix, obfuscation was defined as three criteria: external library calls, reflection, and native code usage.	1. External libraries can contain unexpected and/or malicious functionality. 2. Reflective calls can obscure the control flow of an app and/or subvert permissions within an app. 3. Native code (code written in languages other than Java in Android) can perform unexpected and/or malicious functionality.
Excessive Power Consumption	Excessive functions or unintended apps running on a device which intentionally or unintentionally drain the battery.	Shortened battery life could affect the ability to perform mission-critical functions.

Type	Description	Negative Consequence
Traditional Software Vulnerabilities	All vulnerabilities associated with traditional Java code including: Authentication and Access Control, Buffer Handling, Control Flow Management, Encryption and Randomness, Error Handling, File Handling, Information Leaks, Initialization and Shutdown, Injection, Malicious Logic, Number Handling, and Pointer and Reference Handling.	Common consequences include unexpected outputs, resource exhaustion, denial of service, *etc.*

The table below shows the hierarchy of Android app vulnerabilities from A level to C level.

Table 2. Android Vulnerabilities by Level

Level A	Level B	Level C
Permissions	Over Granting	Over Granting in Code
		Over Granting in API
	Under Granting	Under Granting in Code
		Under Granting in API
	Developer Created Permissions	Developer Created in Code
		Developer Created in API
	Implicit Permission	Granted through API
		Granted through Other Permissions
		Granted through Grandfathering
Exposed Communications	External Communications	Bluetooth
		GPS
		Network/Data Communications
		NFC Access
	Internal Communications	Unprotected Intents
		Unprotected Activities
		Unprotected Services
		Unprotected Content Providers
		Unprotected Broadcast Receivers
		Debug Flag

Potentially Dangerous Functionality	Direct Addressing	Memory Access
		Internet Access
	Potentially Dangerous API	Cost Sensitive APIs
		Personal Information APIs
		Device Management APIs
	Privilege Escalation	Altering File Privileges
		Accessing Super User/Root
App Collusion	Content Provider/Intents	Unprotected Content Providers
		Permission Protected Content Providers
		Pending Intents
	Broadcast Receiver	Broadcast Receiver for Critical Messages
	Data Creation/Changes/Deletion	Creation/Changes/Deletion to File Resources
		Creation/Changes/Deletion to Database Resources
	Number of Services	Excessive Checks for Service State
Obfuscation	Library Calls	Use of Potentially Dangerous Libraries
		Potentially Malicious Libraries Packaged but Not Used
	Native Code Detection	
	Reflection	
	Packed Code	
Excessive Power Consumption	CPU Usage	
	I/O	

Appendix C—iOS App Vulnerability Types

This appendix identifies and defines the various types of vulnerabilities that are specific to apps running on mobile devices utilizing the Apple iOS operating system. The scope does not include vulnerabilities in the mobile platform hardware and communications networks. Although some of the vulnerabilities described below are common across mobile device environments, this appendix focuses on iOS-specific vulnerabilities.

The vulnerabilities in this appendix are broken into three hierarchical levels, A, B, and C. The A level is referred to as the vulnerability class and is the broadest description for the vulnerabilities specified under that level. The B level is referred to as the sub-class and attempts to narrow down the scope of the vulnerability class into a smaller, common group of vulnerabilities. The C level specifies the individual vulnerabilities that have been identified. The purpose of this hierarchy is to guide the reader to finding the type of vulnerability they are looking for as quickly as possible.

The A level general categories of iOS app vulnerabilities are listed below:

Table 3. iOS Vulnerability Descriptions, A Level

Type	Description	Negative Consequence
Privacy	Similar to Android Permissions, iOS privacy settings allow for user-controlled app access to sensitive information. This includes: contacts, Calendar information, tasks, reminders, photos, and Bluetooth access.	iOS lacks the ability to create shared information and protect it. All paths of information sharing are controlled by the iOS app framework and may not be extended. Unlike Android, these permissions may be modified later for individual permissions and apps.
Exposed Communication-Internal and External	Internal communications protocols allow apps to process information and communicate with other apps. External communications allow information to leave the device.	Exposed internal communications allow apps to gather unintended information and inject new information. Exposed external communication (data network, Wi-Fi, Bluetooth, *etc.*) leave information open to disclosure or man-in-the-middle attacks.
Potentially Dangerous Functionality	Controlled functionality that accesses system-critical resources or the user's personal information. This functionality can be invoked through API calls or hard coded into an app.	Unintended functions could be performed outside the scope of the app's functionality.
App Collusion	Two or more apps passing information to each other in order to increase the capabilities of one or both apps beyond their declared scope.	Collusion can allow apps to obtain data that was unintended such as a gaming app obtaining access to the user's contact list.
Obfuscation	Functionality or control flow that is hidden or obscured from the user. For the purposes of this appendix, obfuscation was defined as three criteria: external library calls, reflection, and packed code.	1. External libraries can contain unexpected and/or malicious functionality. 2. Reflective calls can obscure the control flow of an app and/or subvert permissions within an app. 3. Packed code prevents code reverse engineering and can be used to hide malware.
Excessive Power Consumption	Excessive functions or unintended apps running on a device which intentionally or unintentionally drain the battery.	Shortened battery life could affect the ability to perform mission-critical functions.

Type	Description	Negative Consequence
Traditional Software Vulnerabilities	All vulnerabilities associated with Objective C and others. This includes: Authentication and Access Control, Buffer Handling, Control Flow Management, Encryption and Randomness, Error Handling, File Handling, Information Leaks, Initialization and Shutdown, Injection, Malicious Logic, Number Handling and Pointer and Reference Handling.	Common consequences include unexpected outputs, resource exhaustion, denial of service, *etc.*

The table below shows the hierarchy of iOS app vulnerabilities from A level to C level.

Table 4. iOS Vulnerabilities by Level

Level A	Level B	Level C
Privacy	Sensitive Information	Contacts
		Calendar Information
		Tasks
		Reminders
		Photos
		Bluetooth Access
Exposed Communications	External Communications	Telephony
		Bluetooth
		GPS
		SMS/MMS
		Network/Data Communications
	Internal Communications	Abusing Protocol Handlers
Potentially Dangerous Functionality	Direct Memory Mapping	Memory Access
		File System Access
	Potentially Dangerous API	Cost Sensitive APIs
		Device Management APIs
		Personal Information APIs
App Collusion	Data Change	Changes to Shared File Resources
		Changes to Shared Database Resources
		Changes to Shared Content Providers
	Data Creation/Deletion	Creation/Deletion to Shared File Resources
Obfuscation	Number of Services	Excessive Checks for Service State
	Native Code	Potentially Malicious Libraries Packaged but not Used
		Use of Potentially Dangerous Libraries
		Reflection Identification
		Class Introspection
	Library Calls	Constructor Introspection
		Field Introspection
		Method Introspection

	Packed Code	
Excessive Power Consumption	CPU Usage	
	I/O	

Appendix D—Glossary

Selected terms used in this publication are defined below.

Administrator	A member of the organization who is responsible for deploying, maintaining, and securing the organization's mobile devices as well as ensuring that deployed devices and their installed apps conform to the organization's security requirements.
Analyzer	A service, tool, or human that tests an app for specific software vulnerabilities.
App Security Requirement	A requirement that ensures the security of an app. A general requirement is an app security requirement that defines software characteristics or behavior that an app must exhibit to be considered secure. A context-sensitive requirement is an app security requirement that is specific to the organization. An organization's app security requirements comprise a subset of general and context-sensitive requirements.
App Vetting Process	The process of verifying that an app meets an organization's security requirements. An app vetting process comprises app testing and app approval/rejection activities.
Approver	A member of the organization that determines the organization's official approval or rejection of an app.
Auditor	A member of the organization who inspects reports and risk assessments from one or more analyzers as well as organization-specific criteria to ensure that an app meets the security requirements of the organization.
Dynamic Analysis	Detecting software vulnerabilities by executing an app using a set of input use cases and analyzing the app's runtime behavior.
Fault Injection Testing	Attempting to artificially cause an error with an app during execution by forcing it to experience corrupt data or corrupt internal states to see how robust it is against these simulated failures.
Functionality Testing	Verifying that an app's user interface, content, and features perform and display as designed.
Personally Identifiable Information	Any information about an individual that can be used to distinguish or trace an individual's identify and any other information that is linked or linkable to an individual [45].
Risk Assessment	A value that defines an analyzer's estimated level of security risk for using an app. Risk assessments are typically based on the likelihood that a detected vulnerability will be exploited and the impact that the detected vulnerability may have on the app or its related device or network. Risk assessments are typically represented as categories (e.g., low-, moderate-, and high-risk).
Static Analysis	Detecting software vulnerabilities by examining the app source code and binary and attempting to reason over all possible behaviors that might arise at runtime.
Software Assurance	The level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle and that the software functions in the intended manner.
Software Correctness Testing	The process of executing a program with the intent of finding errors and is aimed primarily at improving quality assurance, verifying and validating described functionality, or estimating reliability.
Software Vulnerability	A security flaw, glitch, or weakness found in software that can be exploited by an attacker.

Appendix E—Acronyms and Abbreviations

Selected acronyms and abbreviations used in this publication are defined below.

3G	3rd Generation
API	Application Programming Interface
CAPEC	Common Attack Pattern Enumeration and Classification
CMVP	Cryptographic Module Validation Program
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DHS	Department of Homeland Security
DoD	Department of Defense
DOJ	Department of Justice
EULA	End User License Agreement
FIPS	Federal Information Processing Standard
FISMA	Federal Information Security Management Act
GPS	Global Positioning System
I/O	Input/Output
IT	Information Technology
ITL	Information Technology Laboratory
JVM	Java Virtual Machine
LTE	Long-Term Evolution
MDM	Mobile Device Management
NFC	Near Field Communications
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
OMB	Office of Management and Budget
OS	Operating System
PII	Personally Identifiable Information
QoS	Quality of Service
SAMATE	Software Assurance Metrics And Tool Evaluation
SCAP	Security Content Automation Protocol
SD	Secure Digital
SDK	Software Development Kit
SLA	Service-Level Agreement
SMS	Short Message Service
SP	Special Publication
UI	User Interface
VPN	Virtual Private Network
Wi-Fi	Wireless Fidelity

Appendix F—References

References for this publication are listed below.

[1] Committee on National Security Systems (CNSS) Instruction 4009, *National Information Assurance Glossary*, April 2010. https://www.cnss.gov/CNSS/issuances/Instructions.cfm (accessed 12/4/14).

[2] National Aeronautics and Space Administration (NASA) NASA-STD-8739.8 w/Change 1, *Standard for Software Assurance*, July 28, 2004 (revised May 5, 2005). http://www.hq.nasa.gov/office/codeq/doctree/87398.htm (accessed 12/4/14).

[3] Radio Technical Commission for Aeronautics (RTCA), RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*, December 1, 1992 (errata issued March 26, 1999).

[4] International Electrotechnical Commission (IEC), IEC 61508, *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*, edition 2.0 (7 parts), April 2010.

[5] International Organization for Standardization (ISO), ISO 26262, *Road vehicles -- Functional safety* (10 parts), 2011.

[6] M. Souppaya and K. Scarfone, *Guidelines for Managing the Security of Mobile Devices in the Enterprise,* NIST Special Publication (SP) 800-124 Revision 1, June 2013. http://dx.doi.org/10.6028/NIST.SP.800-124r1.

[7] National Information Assurance Partnership, *Protection Profile for Mobile Device Fundamentals Version 2.0*, September 17, 2014. https://www.niap-ccevs.org/pp/PP_MD_v2.0/ (accessed 12/4/14).

[8] Joint Task Force Transformation Initiative, *Security and Privacy Controls for Federal Information Systems and Organizations*, NIST Special Publication (SP) 800-53 Revision 4, April 2013 (updated January 15, 2014). http://dx.doi.org/10.6028/NIST.SP.800-53r4.

[9] NIST, *AppVet Mobile App Vetting System* [Web site], http://csrc.nist.gov/projects/appvet/ (accessed 12/4/14).

[10] Joint Task Force Transformation Initiative, *Guide for Conducting Risk Assessments*, NIST Special Publication (SP) 800-30 Revision 1, September 2012. http://csrc.nist.gov/publications/nistpubs/800-30-rev1/sp800_30_r1.pdf (accessed 12/4/14).

[11] G. McGraw, *Software Security: Building Security In*, Upper Saddle River, New Jersey: Addison-Wesley Professional, 2006.

[12] M. Dowd, J. McDonald and J. Schuh, *The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities*, Upper Saddle River, New Jersey: Addison-Wesley Professional, 2006.

[13] H.G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society* 74, pp. 358-366, 1953. http://dx.doi.org/10.1090/S0002-9947-1953-0053041-6.

[14] J.H. Allen, S. Barnum, R.J. Ellison, G. McGraw, and N.R. Mead, *Software Security Engineering: a Guide for Project Managers*, Upper Saddle River, New Jersey: Addison-Wesley, 2008.

[15] Microsoft Corporation, *The STRIDE Threat Model* [Web site], http://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx, 2005 (accessed 12/4/14).

[16] *Trike* - open source threat modeling methodology and tool [Web site], http://www.octotrike.org (accessed 12/4/14).

[17] K. Scarfone, M. Souppaya, A. Cody and A. Orebaugh, *Technical Guide to Information Security Testing and Assessment*, NIST Special Publication (SP) 800-115, September 2008. http://csrc.nist.gov/publications/nistpubs/800-115/SP800-115.pdf (accessed 12/4/14).

[18] D. C. Gause and G. M. Weinberg, *Exploring Requirements: Quality Before Design*, New York: Dorset House Publishing, 1989.

[19] B. Nuseibeh and S. Easterbrook, "Requirements engineering: a roadmap," *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, Limerick, Ireland, June 7-9, 2000, New York: ACM, pp. 35-46. http://dx.doi.org/10.1145/336512.336523.

[20] NIST, *Cryptographic Module Validation Program (CMVP)* [Web site], http://csrc.nist.gov/groups/STM/cmvp/ (accessed 12/4/14).

[21] E. Barker and Q. Dang, *Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance*, NIST Special Publication (SP) 800-57 Part 3 Revision 1 (DRAFT), May 2014. http://csrc.nist.gov/publications/drafts/800-57pt3_r1/sp800_57_pt3_r1_draft.pdf (accessed 12/4/14).

[22] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*, Hoboken, New Jersey: John Wiley & Sons, Inc., 2008.

[23] G.G. Schulmeyer, ed., *Handbook of Software Quality Assurance*, 4th Edition, Norwood, Massachusetts: Artech House, Inc., 2008.

[24] B.B. Agarwal, S.P. Tayal and M. Gupta, *Software Engineering and Testing*, Sudbury, Massachusetts: Jones and Bartlett Publishers, 2010.

[25] J.R. Maximoff, M.D. Trela, D.R. Kuhn and R. Kacker, *A Method for Analyzing System State-space Coverage within a t-Wise Testing Framework, 4th Annual IEEE International Systems Conference*, April 5-8, 2010, San Diego, California, pp. 598-603. http://dx.doi.org/10.1109/SYSTEMS.2010.5482481.

[26] G.J. Myers, *The Art of Software Testing*, second edition, Hoboken, New Jersey: John Wiley & Sons, Inc., 2004.

[27] OW2 Consortium, *ASM* - Java bytecode manipulation and analysis framework [Web site], http://asm.ow2.org/ (accessed 12/4/14).

[28] H. Chen, T. Zou and D. Wang, "Data-flow based vulnerability analysis and java bytecode," *7th WSEAS International Conference on Applied Computer Science (ACS'07)*, Venice, Italy, November 21-23, 2007, pp. 201-207. http://www.wseas.us/e-library/conferences/2007venice/papers/570-602.pdf (accessed 12/4/14).

[29] University of Maryland, *FindBugs: Find Bugs in Java Programs* - Static analysis to look for bugs in Java code [Web site], http://findbugs.sourceforge.net/ (accessed 12/4/14).

[30] R.A. Shah, *Vulnerability Assessment of Java Bytecode*, Master's Thesis, Auburn University, December 16, 2005. http://hdl.handle.net/10415/203.

[31] G. Zhao, H. Chen and D. Wang, "Data-Flow Based Analysis of Java Bytecode Vulnerability," In *Proceedings of the Ninth International Conference on Web-Age Information Management (WAIM '08)*, Zhanjiajie, China, July 20-22, 2008, Washington, DC: IEEE Computer Society, pp. 647-653. http://doi.ieeecomputersociety.org/10.1109/WAIM.2008.99.

[32] G. Balakrishnan, *WYSINWYX: What You See Is Not What You eXecute*, Ph.D. diss. [and Tech. Rep. TR-1603], Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin, August 2007. http://research.cs.wisc.edu/wpis/papers/balakrishnan_thesis.pdf (accessed 12/4/14).

[33] NIST, *SAMATE: Software Assurance Metrics And Tool Evaluation* [Web site], http://samate.nist.gov/Main_Page.html (accessed 12/4/14).

[34] The MITRE Corporation, *Common Weakness Enumeration CWE-190: Integer Overflow or Wraparound* [Web site], http://cwe.mitre.org/data/definitions/190.html (accessed 12/4/14).

[35] The MITRE Corporation, *Common Weakness Enumeration CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')* [Web site], http://cwe.mitre.org/data/definitions/77.html (accessed 12/4/14).

[36] NIST, *National Vulnerability Database (NVD)* [Web site], http://nvd.nist.gov/ (accessed 12/4/14).

[37] NIST, *Security Content Automation Protocol (SCAP)* [Web site], http://scap.nist.gov/ (accessed 12/4/14).

[38] The MITRE Corporation, *Common Weakness Enumeration* [Web site], http://cwe.mitre.org/ (accessed 12/4/14).

[39] The MITRE Corporation, *Common Attack Pattern Enumeration and Classification* [Web site], https://capec.mitre.org/ (accessed 12/4/14).

[40] R.A. Martin and S.M. Christey, "The Software Industry's 'Clean Water Act' Alternative," *IEEE Security & Privacy*10(3), pp. 24-31, May-June 2012. http://dx.doi.org/10.1109/MSP.2012.3.

[41] Forum of Incident Response and Security Teams (FIRST), *Common Vulnerability Scoring System (CVSS-SIG)* [Web site], http://www.first.org/cvss (accessed 12/4/14).

[42] Industry Consortium for Advancement of Security on the Internet (ICASI), *The Common Vulnerability Reporting Framework (CVRF)* [Web site], http://www.icasi.org/cvrf (accessed 12/4/14).

[43] U.S. Department of Homeland Security, *Software & Supply Chain Assurance – Community Resources and Information Clearinghouse (CRIC): Questionnaires* [Web site], https://buildsecurityin.us-cert.gov/swa/forums-and-working-groups/acquisition-and-outsourcing/resources#ques (accessed 12/4/14).

[44] The MITRE Corporation, *Common Vulnerabilities and Exposures (CVE)* [Web site], https://cve.mitre.org/ (accessed 12/4/14).

[45] E. McCallister, T. Grance and K. Scarfone, *Guide to Protecting the Confidentiality of Personally Identifiable Information (PII)*, NIST Special Publication (SP) 800-122, April 2010. http://csrc.nist.gov/publications/nistpubs/800-122/sp800-122.pdf (accessed 12/4/14).