Aug 19, 2014

SP 800-163

DRAFT Technical Considerations for Vetting 3rd Party Mobile Applications

NIST announces that Draft Special Publication 800-163, *Technical Considerations for Vetting 3rd Party Mobile Applications*, is now available for public comment. The purpose of this document is to provide guidance for vetting 3rd party software applications (apps) for mobile devices. Mobile app vetting is intended to assess a mobile app's operational characteristics of secure behavior and reliability (including performance) so that organizations can determine if the app is acceptable for use in their expected environment. This document provides key technical software assurance considerations for organizations as they adopt mobile app vetting processes.

NIST requests comments on Draft Special Publication 800-163 by 5:00pm EDT on ***September 18, 2014***. Please submit comments using the SP 800-163 comment template (see link below for Excel spreadsheet) to nist800-163 @ nist.gov with "Comments on Draft SP 800-163" in the subject line.

NIST Special Publication 800-163 (Draft)

Technical Considerations for Vetting 3rd Party Mobile Applications (Draft)

Jeffrey Voas
Steve Quirolgico
Christoph Michael
Karen Scarfone

COMPUTER SECURITY

NIST Special Publication 800-163

Technical Considerations for Vetting 3rd Party Mobile Applications (Draft)

Jeffrey Voas, Steve Quirolgico
Computer Security Division
Information Technology Laboratory

Christoph Michael
Leidos

Karen Scarfone
Scarfone Cybersecurity

August 2014

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

National Institute of Standards and Technology Special Publication 800-163
Natl. Inst. Stand. Technol. Spec. Publ. 800-163, 45 pages (August 2014)
CODEN: NSPUE2

This publication is available free of charge from:

Comments on this publication may be submitted to:

Public comment period: *August 18, 2014* through *September 18, 2014*

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

Reports on Computer Systems Technology

Abstract

Today's commercially available mobile devices (e.g., smartphones, tablets) are handheld computing platforms with wireless capabilities, geographic localization, cameras, and microphones. Similar to computing platforms such as desktops and laptops, the user experience with a mobile device is tied to the software apps and the tools and utilities available. The purpose of this document is to provide guidance for vetting 3rd party software applications (apps) for mobile devices. Mobile app vetting is intended to assess a mobile app's operational characteristics of secure behavior and reliability (including performance) so that organizations can determine if the app is acceptable for use in their expected environment.

Keywords

malware; mobile apps; mobile devices; smartphones; software reliability; software security; software testing; software vetting;

Acknowledgments

Special thanks to the National Security Agency's Center for Assured Software for providing Appendices C and D which tabulate platform-specific mobile app vulnerabilities.

TBD

Trademarks

Table of Contents

Table of Figures and Tables

Executive Summary

Organizations should develop or adopt requirements that they expect mobile software applications (apps) they will use on their organization's mobile platforms to meet. These requirements should reflect the organization's unique mission requirements, understanding of their IT infrastructure, choices of mobile devices and configurations, and the organization's acceptable levels of risk. The process of verifying that a software application meets those requirements is known as *vetting*. Vetting can be applied to either binary code or source code and the code can be examined as static artifacts or the binary can be examined while running dynamically. This document provides technical considerations for vetting 3rd party mobile applications that are commonly known as *mobile apps*. Emulated and virtual environments are useful for vetting mobile apps because they allow having many different test configurations (device, OS version, app version, and system parameters) when examining the running binary dynamically and restoring particular states or the entire hardware and system configuration, without altering an actual mobile device. They allow for automated testing and examination as well as exploration of the app by skilled human analysts.

This document provides key technical considerations for organizations as they adopt mobile app vetting processes. The following are key recommendations made within this publication:

Understand the security and privacy risks mobile apps present and have a strategy for mitigating them.

When deploying a new technology, organizations should be aware of the potential security and privacy impact these technologies may have on the organization's IT resources, data, and users. New technologies may offer the promise of productivity gains and new capabilities, but if these new technologies present new risks, the organization's IT professionals, users, and business owners should be fully aware of these new risks and develop plans to mitigate them or be fully informed before accepting the consequences of them. Federal, State, Local and Tribal privacy statutes may be different for each organization; security administrators should consult with the organization's privacy officer or legal counsel to ensure the collection and sharing of data collected using mobile devices is legal.

Provide mobile app security and privacy training for your employees.

Employers should ensure that their employees understand the organization's mobile device use policies and how mobile apps may compromise the organization's security and the user's privacy. Users should be educated about Personally Identifiable Information (PII), how PII can be accessed and shared, how the privacy- and security-relevant capabilities of a mobile device relate to PII, as well as how an organization's resources can be protected or put at risk by the user's actions and inactions. Training programs should describe how mobile devices can collect information about the user and how that information can be shared with third parties through the apps running on the mobile device.

To provide long-term assurance of the software throughout its lifecycle, all mobile apps, as well as their updates, should go through a mobile app vetting process.

Each new version of a software application can potentially introduce new, unintentional weaknesses or unreliable code into an existing mobile app. New versions of mobile apps should be treated as new mobile apps and should go through an app vetting process. For many organizations this software assurance step does not currently take place for mobile apps. This is in contrast to new releases of desktop or server enterprise software, which are typically tested in a controlled setting or in a staging area before being deployed across an enterprise. This traditional enterprise software assurance model, however, does not scale well to meet user demands for the wide array of apps available for mobile devices Mobile apps

developed in-house should also be tested and analyzed for security and reliability before being released to the community of users within the organization. The purpose of these tests and the analysis is to validate that an app adheres to a predefined acceptable level of security risk and to identify whether the developers have introduced any latent weaknesses that could make the IT infrastructure vulnerable or expose the user's confidential data.

The mobile app update model can help mitigate risks when used properly. Security administrators should establish a process for quickly vetting security-related app updates.

Mobile app update notifications can be sent directly to the user's device and downloaded from an app store or marketplace and bypass the traditional IT testing, staging, approval, and deployment process. This model offers the promise of pushing critical security updates faster to the end user, but it also introduces the risk of having unvetted code be installed on a user's device with unknown consequences and security implications. The first release of a mobile app may be free of any malware, but malicious code may be introduced in subsequent updates. Security administrators should establish a process for quickly vetting security-related app updates.

Stakeholders should be made aware of what the mobile app vetting process does and does not provide in terms of secure behavior of apps.

A mobile apps vetting system is comprised of a set of tools and methodologies for addressing specified app vetting requirements. Each mobile app vetting tool or methodology generates results that identify security, privacy, reliability, accessibility, and performance issues with the mobile app. Any vetting approach should be repeatable, efficient, consistent, and limit errors (e.g., false positives, false negatives). As with any software assurance process, there is no guarantee that even the most thorough vetting processes will uncover all vulnerabilities. Stakeholders should be made aware that although app security assessments should generally improve the security posture of an organization, the degree to which it does so may not be easily or immediately ascertained. Stakeholders should also be educated on the value of humans in security assessment processes. Security analysis is primarily a human-driven process that can be highly augmented and aided by automated tools, but the tool results themselves need to be evaluated in the context of the intended mission/business use of the app and the capabilities and configuration of the mobile platform and networking setup. There is a residual risk associated with any capability, including mobile apps. App security assessments are intended to reduce that level of risk by identifying and mitigating vulnerabilities. The collection and analysis of metrics can indicate the level of risk reduction, but it is impossible to completely nullify all risk.

Mobile apps are part of a larger system; mobile app testing results should be reviewed by a software analyst within the context of an organization's mission objectives, security posture, and risk tolerance.

End user's mobile devices, data, and network resources face multiple threats that can originate from many places and that depend on the sophistication of the adversary. Some risks can be mitigated through user education and operational procedures, while other risks can only be mitigated through the use of technical countermeasures. Mobile app deficiencies detected through the mobile app vetting process may be mitigated by technical countermeasures that are part of a larger system. Mobile apps that meet minimum technical requirements may not be suitable for an organization to use because they do not meet acceptable use policies. Automated testing is necessary for the vetting process to scale and keep up with the demands of the user base, but the manual review of the test results by a human analyst will best serve an organization's software assurance goals.

1. Introduction

1.1 Purpose and Scope

The purpose of this document is to provide guidance for vetting 3rd party software applications (apps) for mobile devices. Mobile app vetting is intended to assess a mobile app's operational characteristics of secure behavior and reliability (including performance) so that organizations can determine if the app is acceptable for use in their expected environment[1]. This document is not a step-by-step guide for performing software vetting, but rather highlights those elements that are particularly important to be considered before mobile apps are approved as "fit-for-use." This document does not address the robustness, security, or reliability of the underlying mobile platform and operating system, which are addressed in other publications, such as the Protection Profile for Mobile Device Fundamentals[2]. While these are important characteristics for organizations to understand and consider in selecting mobile devices, this document is focused on how to vet mobile apps after the choice of platform has been made. Ultimately, the acceptance of a mobile app depends on the environment in which it is deployed, the context in which it will be used, and the underlying security technologies supporting the use of mobile apps. Organizations developing apps in-house should also refer to guidance on secure programming techniques and software quality assurance processes to appropriately address the entire software development lifecycle [MCGRAW05, SCHUL07].

1.2 Audience

This document is intended for individuals or organizations that will be vetting, assessing, and acquiring mobile apps, as well as those with responsibilities for setting app vetting policies and practices. Mobile app developers may also be interested in this document for its survey of issues they should consider when developing and distributing mobile apps.

1.3 Document Structure

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

- Section 2 provides a brief overview of software assurance issues for mobile apps.
- Section 3 provides guidance for organizations planning to set up an in-house mobile app vetting process.
- Section 4 discusses common mobile app testing requirements, such as security, privacy, functionality, performance, and reliability.
- Section 5 examines mobile app vetting tools and techniques.
- Appendix A discusses power consumption testing for apps.
- Appendices B and C identify and define platform-specific vulnerabilities for mobile apps running on the Android and iOS operating systems, respectively.

[1] For purposes of this document, reliability measures how likely a function will be present at a point in time and operate as desired, and performance measures its speed of execution.

[2] See Protection Profile for Mobile Device Fundamentals Version 1.1 https://www.niap-ccevs.org/pp/PP_MD_v1.1/

- Appendix D defines selected terms used in the document.
- Appendix E defines selected acronyms and abbreviations used in the document.
- Appendix F lists references for the publication.

2. Software Assurance for Mobile Apps

This section discusses the challenges of software assurance in mobile computing and software assurance in apps. *Software assurance* can be defined as “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.” [CNSS-4009] 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.” [NASA-8739] There are a number of government and industry legacy software assurance standards that are primarily directed at the process for developing applications that require a high level of assurance: for example, space flight, automotive systems, and critical defense systems.[3]

Although considerable progress has been made in the past decades in the area of software assurance, and considerable 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 the model under which mobile applications are developed and used does not necessarily support traditional software assurance techniques. At the same time, mobile apps are smaller and more addressable by the static and dynamic analysis techniques available then the large applications used in traditional IT platforms. This document discusses how app vetting might be used to further improve the secure behavior and correctness of mobile apps.

2.1 Challenges of Software Assurance in Mobile Computing

The rapidly-evolving mobile app marketplace economic model challenges the traditional software development process in a number of ways. Mobile app developers are attracted by the opportunities to reach a market of millions of users overnight. However they may have little experience building quality software that is reliable and 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 free trial versions that may contain functionality flaws and/or security and resilience-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 this free or inexpensive software often tolerate buggy, unreliable, and insecure code in exchange for the low cost. Mobile app developers typically update their apps much more frequently than traditional applications. This “open-ended” development cycle can foster dangerous behaviors when it comes to the reliability and secure behavior of the application. At the same time, the mobile app deployment model offers the opportunity to ensure security patches are widely and rapidly deployed.

Enterprises 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. They are also finding that an increasing amount of their business processes are conducted on employee-owned mobile devices. This contrasts with the traditional information infrastructure, where enterprises support approved desktop applications and the average employee only uses a handful of applications and web-based enterprise databases to do the majority of their work. Mobile devices provide access to potentially

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

millions of mobile apps for a user to choose from. This trend challenges the traditional mechanisms of enterprise IT security, which relied on a tightly controlled environment, where the software available to employees was uniform throughout the organization and communication was filtered using corporate firewalls and common endpoint configurations.

Like traditional software applications, mobile apps also suffer from other issues. For example, some apps are not entirely self-contained, but instead rely on third-party libraries that can be closed source, self-modifying code, or that can execute unknown server-side code for additional functionality. These third-party libraries, over which a developer has no control, can contain malware and can have vulnerabilities at the time of use or that are found later and become an operational risk until patched.

A major difference between mobile apps and enterprise applications is that unlike a desktop computing system, far more precise and continuous device location information, physical sensor data, and pictures and audio about a user can be exposed to third-party app developers or third-party services. Mobile apps can sense and store information including user Personally Identifiable Information (PII) data through mobile device services. Although many mobile 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. Therefore, mobile app vetting must also include identifying potential privacy issues.

Unlike traditional laptop computers, mobile devices have access to a wide variety of network services, including Wi-Fi and 2G/3G and 4G/Long Term Evolution (LTE). This is in addition to the short-range data connectivity provided by services such as Bluetooth, and Near Field Communications (NFC). All these avenues of data transmission are typically available to the apps that run on a mobile device and many have been shown to be potential vectors for remote exploits. Mobile devices cannot be physically protected in an agency's buildings and therefore an adversary can much more easily get their hands on a lost or stolen mobile device.

2.2 Software Assurance for Mobile Apps

The software distribution model that has arisen with mobile computing presents challenges to software assurance analysts, but it also creates opportunities. The mobile app vetting process acknowledges the concept that someone other than the software vendor is entitled to evaluate the software's reliability and secure behavior. This may make it easier for organizations to evaluate software in the context of their own security policies, planned use, and risk tolerance, especially if they are willing to set up their own app vetting capability. But distancing the developer from software assurance activities can also make those activities less effective with respect to improving secure behavior and reliability.

To provide long-term assurance of the software throughout its lifecycle, all apps, as well as their updates, should go through a software assurance vetting process, because each new version of a software application can introduce new unintentional weaknesses or unreliable code.

Mobile apps should be tested for secure behavior and reliability before being released to the community of users within the organization. The purpose of these tests 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.

Organizations will need to define their own acceptance criteria for mobile app tests to address their unique mission requirements, platform, configuration, and threat environment. Acceptance criteria refer to the evidence provided that demonstrates that a mobile app meets the reliability and security expectations of the final approver. A military hospital may use mobile device sensors to allow doctors to record observations or instructions related to patient medical care while other organizations may prohibit the use of sensing devices in sensitive areas. Some organizations may encourage the use of social networking apps as part of their outreach programs to the public while other organizations may prohibit the use of any apps that profile their users. Some risks may be mitigated by security services offered through Mobile Device Management (MDM) technologies while other risks may be mitigated by the use of sandboxing or other commercially available technologies. Enterprise administrators can make use of MDM software that can provide access to administrative functions and regulate some of the mobile device functionality. One such feature is the ability of an MDM to allow users to download apps that are on lists of preapproved apps and for the organization to keep track of apps and version numbers installed on the organization's managed mobile devices. The MDM can respond to a report of a vulnerability and release a patched version to address the risk to the organization. However, using such technologies to decide which apps to allow relies on a mobile app approval process and some method for deciding which apps to approve. That approval process should be based on some measurable software assurance properties and enterprise IT policy criteria.

Apps should be used within a certain security context; conducting risk analysis before deploying mobile apps should identify the roles of users, the risks, and the available countermeasures (See NIST SP 800-124, *Guidelines for Managing the Security of Mobile Devices in the Enterprise*, for more information).[4]

Application software assurance activity costs should be included in project budgets and should not be an afterthought. Organizations that are hiring contractors to develop mobile apps should specify that mobile app assessment costs for static, dynamic, penetration testing, and directed misuse and abuse testing be included as part of the development process. For apps developed in-house, attempting to implement mobile app assessments solely at the end of the development effort will lead to increased costs, lengthened project timelines, and poorer quality apps reaching the user community. It is essential to identify any potential vulnerabilities or weaknesses during the development process when they can still be addressed by the original developers without leading to a cascade effect across the rest of the app.

Because mobile apps and their behaviors are context-sensitive, it may be the case that individual organizations will need to conduct their own vetting and testing processes. It is doubtful that given the current state of the technology a "one size fits all" approach to app vetting is plausible but the basic findings from app vetting may be reusable by others. Sharing an organization's findings with others or leveraging another organization's findings for an app should be contemplated to avoid duplicating work and wasting scarce app vetting capabilities. With appropriate standards for scoping, managing, licensing, and recording the findings from assurance activities in consistent ways, the different activities can possibly be looked at collectively and common problems and solutions may be applicable across the industry or with similar organizations.

[4] http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-124r1.pdf

3. Mobile App Vetting Planning

When deploying a new technology, organizations should be aware of the potential security and privacy impact these technologies may have on the organization's IT resources, data, and users. New technologies may offer the promise of productivity gains and new capabilities, but if these new technologies present new risks, the organization's IT professionals and users should be fully aware of these new risks and develop plans to mitigate them or be fully informed before accepting the consequences of them. Organizations should develop mobile app testing requirements to ensure that mobile apps comply with their organization's policies. The process of verifying that an app meets those requirements is known as *vetting*. App stores may perform some app vetting processes to verify compliance with their own requirements; however, because each app store has its own unique vetting processes and requirements which are not always transparent, it is necessary to consult the current agreements and documentation for a particular app store to get more precise information about its practices. Organizations should not assume that an app has been fully vetted to their organizational needs just because it is available through an official app store. Mobile Device Management (MDM), Mobile Application Management (MAM), or other technologies may also include mobile app vetting as a precondition for allowing users to download mobile apps onto their devices. Moreover, 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 assurance.

Organizations should assess the potential risk introduced by use of each app and perform the necessary degree of vetting to ensure that the risk is properly addressed. It is noted that some types of vulnerabilities or weaknesses discovered in mobile apps may be mitigated by other security controls included in the enterprise mobile device architecture.

3.1 Mobile App Vetting Planning

Any vetting approach should be repeatable, efficient, consistent, and limit errors (e.g., false positives, false negatives). It may be effective to initially gear assessment processes to the specific policies of the organization and should not just be an embodiment of the organization's existing security policy. Mobile app vetting process presents the opportunity to improve the organization's security capabilities rather than merely perpetuating it.

Additional considerations for mobile app evaluations include the following:

- Do the stakeholders have specific security 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? All of these questions will have to be explored in detail once the mobile app vetting process is operational, but knowing the stakeholders' concerns can help establish a more effective vetting process.
- Are the critical assets located on mobile devices, or is the concern simply that mobile devices will be used as a springboard to attack those assets?
- Are threat assessments being performed?
- What characteristics of attacks and attackers are the stakeholders 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?
- Are the stakeholders only interested in vetting apps, or is a mobile app vetting process expected to evaluate other aspects of mobile device security such as the OS, firmware, hardware, and communications? Is the mobile app vetting process only meant 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 stakeholders expect more than this the vetting system designers should be aware of it.
- How many apps per week will the mobile application 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 stakeholders 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.

3.2 Existing Security Infrastructure

A mobile apps vetting system should be part of the stakeholders’ overall security strategy. For example, if the stakeholders can use detailed information about types of software vulnerabilities, or weaknesses, to perform in-depth risk assessments, a mobile app vetting system should be able to provide such information. On the other hand, if the stakeholder cannot evaluate security risks, a mobile app vetting system may have to be prepared to do so, since in some cases there may have to be a tradeoff between the risks and benefits of deploying specific apps. If the stakeholders have a formal process for establishing and documenting security requirements, the mobile app vetting system should be able to consume requirements created by that process. Bug and incident reports created by the stakeholder’s organization(s) might also be consumed by a mobile application vetting system as could public advisories about vulnerabilities in commercial and open source apps as well as in libraries used in app development, and conversely evaluation results from mobile app assessments might have to be consumed by the organization’s bug tracking system.

3.3 Expectation Management

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

Stakeholders should also be educated on the value of humans in security assessment processes and ensure that their app vetting doesn’t rely solely on automated tests. Security analysis is primarily a human-driven process (see [DOWD06, MCGRAW05] for example); 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[5], and in fact current technology has not even reached the limits of what *is* theoretically possible. Complex, multifaceted software architectures cannot be 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 stored in the cloud is a security issue if the data is company sensitive, but not if the data consists of employee-owned digital pictures. Even if the security requirements for a mobile 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, it is a general dictum that security analysis requires humans in the loop, and by extension the quality of the outcome depends, among other things, on the amount of human effort available for an evaluation. Software assurance analysts should be familiar with standard processes and best practices for software security assessment (for example, see [MCGRAW05, STRIDE, TRIKE, BSI]). A robust mobile 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.

Relying solely on app vetting for security instead of planning to build security into a mobile architecture inherently violates a fundamental best practice for software security, which is to include security considerations from the very start of the software development lifecycle [MCGRAW05] The final decision of whether a particular app will be used in an organization must include consideration of the mobile platforms that will host the apps, the configuration of those mobile platforms, the intended mission/business being conducted/supported by the apps, and the security capabilities of the networks and connections needed by the app.

3.4 Getting Started

At an absolute minimum, vetting an app requires access to the app's binary code and the most basic metadata for the app, such as the primary point of contact who can answer questions regarding the app's intended use within the organization. However, vetting can be easier to perform and produce more accurate results if source code is available and if additional information about the app is collected before the vetting evaluation begins. This section describes the information that may be collected to help better assess a mobile app. Note that in many cases, this additional information will only be available if the app was developed on behalf of the organization, for example by internal developers or contractors.

Subsequent to vetting the of the mobile app itself, a human analyst will need to evaluate the test results within the context of the app's planned use, the mobile platforms it will be deployed on, the configurations and capabilities of those platforms when deployed, and the capabilities of the rest of the mobile infrastructure in use by the organization before a determination of whether the organization can accept the residual risks that the app would place on the organization.

Information that is most often deemed helpful for deciding whether or not a mobile app is acceptable for use within an organization's own context includes:

[5] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems." Transactions of the American Mathematical Society 75, 358-366, 1953

- **Requirements**: Collect all the pertinent requirements, security policies, privacy policies, acceptable use policies, and social media guidelines that are applicable to your organization.

- **Provenance**: Identity of the developer, developer's organization, developer's reputation, date received, marketplace/app store consumer reviews, etc.

- **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.[6]

- **Code Components:** Access to the source code provides the ability to conduct additional review not possible with app binaries. Software libraries[7]**,** scripts, images, debugging information, configuration files, and compilation instructions will allow more in depth testing than simply collecting binary code.

- **Mobile App Documentation:**

  - **User Guide:** The mobile 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 testing 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 their internal test effort.

  - **Testing results:** Code review results, penetration test results, and other testing results will indicate what reliability, security, and development 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 actual coding of the app.

  - **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 mobile 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

[6] 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.

[7] App vetting might require the vetting of other components with which the app communicates, such as a third-party library, device, or server. Such vetting is typically difficult because licensing issues can restrict access, and user privileges can restrict access to an app's associated server.

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 DHS Custom Software Questionnaire[8] 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 API?"

3.5 Sharing Software Assurance Information

Information sharing within the software assurance community is vital and can help analysts benefit from the collective efforts of security professionals around the world. The National Vulnerability Database (NVD) is the U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP). This data enables automation of vulnerability management, security measurement, and compliance. NVD includes databases of security checklists, security related software flaws, misconfigurations, product names, and impact metrics. The 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 multi-purpose 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 Common Weakness Enumeration (CWE) and Common Attack Pattern Enumeration and Classification (CAPEC) 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. Semantic Templates and Software Fault Patterns are evolving approaches to standardizing the description of software flaws.

In some cases, agencies will need to use multiple app vetting tools when a single app vetting tool may not meet all their testing requirements or be available for all platforms. Government agencies that are likely to be testing the same apps should discuss licensing agreements with automated mobile app test tool vendors that allow for the sharing of test results. Sharing and comparing machine readable test results can benefit software assurance analysts and help the software assurance industry build new automation tools and services.

4. Mobile App Evaluation

Mobile app requirements state an organization's expectations for app behavior and drive the evaluation process. The policies and goals of the organization can help drive and customize app assessment. When possible, tailoring app assessments to the organization's security, privacy, and acceptable use policies of the organization can help drive and customize mobile app assessments. When possible, tailoring app assessments to the organization's unique mission requirements, risk tolerance, and available security countermeasures is prudent within the overall context of security and reliability concerns and more cost effective in time and resources. Such assessments could minimize the risk of an attacker exploiting behaviors that a mobile app vetting system did not know were insecure since it could not know how the

[8] https://buildsecurityin.us-cert.gov/swa/downloads/CustomSoftwareQuestionnaire.doc

app would be deployed. Unfortunately, it is not always possible to get security requirements from the app end users; the user base may be too broad, the user base may not be able to state their security requirements concretely enough for testing, or they might not have articulated any security requirements or expectations of secure and reliable behavior. (In the two latter cases a skilled and well-resourced software analyst may be able to extract requirements from the users anyway, but it is not to be expected that all mobile application vetting processes will have the means to do this.)

A software analyst may be able to anticipate end user expectations successfully based on what the app claims to do and common notions about what constitutes insecure behavior of software. For example, there may be architectural weaknesses that are commonly found in applications of this sort. The resulting requirements should be taken into account even when end users have supplied specific security policies and descriptions of the intended secure behavior. Examples of mobile app test requirements include:

- **Protect Sensitive Data.** Mobile apps collect, store, and transmit sensitive data whether it is user-generated content (text, photographs, video, email, etc.), location information, data collected by on-board physical sensors, and data shared through network communications. Mobile apps should protect sensitive data at rest and in transit by using cryptographic security services provided by the underlying platform.
- **Preserve Privacy.** The app must properly use any personal information it interacts with—such as asking permission to use personal information and using it only for authorized purposes. This includes location services, geotagging photographs, accessing the microphone and the camera, and other data gather by physical sensors..
- **Perform Basic Functionality.** The app must work as described; all buttons, menu items, and other interfaces must work. When a service or function is unavailable (disabled, unreachable, etc.), the error condition must be handled gracefully.
- **Performance.** The app's launch time and suspend time must be reasonable. The app must meet requirements regarding the usage of metered networks, such as cellular networks (e.g., restrictions on using certain protocols over such networks; data rate restrictions; download size restrictions; asking permission to use cellular network instead of Wi-Fi).
- **Power Consumption.** The app must not consume power in excess of a user- or agency-defined rate consistent. This predefined rate may depend on the underlying hardware and the environment in which the app will be used. For example, a first responder may need to have network and GPS services available for a minimum time while using a set of critical apps.
- **Reliability/Availability.** The app must not crash or hang; it must not enter an endless loop, where the user cannot terminate it in an orderly way. Behavior that strikes the evaluator as inconsistent with the implied or documented purpose of the app will probably seem incorrect to the user as well; in the absence of clear and unambiguous documentation stating otherwise, the evaluator is justified in reporting this as a bug.
- **Security Functionality.** The app only uses designated APIs (from the vendor-provided software development kit [SDK]) and uses them properly; no other API calls are permitted.[9] The app has

[9] 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.

strong access control lists (ACLs) for its executable code, directories/folders, and other objects. Static analysis performed against the binary must not detect the use of any unsafe coding or building practices. The app must also not share memory locations with other apps, possess OS privileges, or contain hard-coded passwords.

- **Content.** The app must have appropriate content, for example, no copyright or trademark infringement, and no offensive material.

Well-written security requirements are most useful to the software analyst 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 (c.f., [GAU89, NUS00]), 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[10] 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.

4.1 Identifying Undesirable App Characteristics

This section discusses security and privacy-relevant characteristics of mobile apps that are generally undesirable: insecure, malicious functionality contains malware, vulnerable, and poorly designed. The mobile app vetting process attempts to identify mobile apps with these characteristics.

Insufficient Data Protection. Sensitive data stored on the device and transmitted over one of many wireless interface now available on mobile device should be protected from eavesdropping, from other mobile apps, and even in the case of a lost or stolen device. When implemented properly, the use of cryptography can help maintain the confidentiality and integrity of sensitive data. FIPS 140-2 precludes the use of unvalidated cryptography for the cryptographic protection of sensitive or valuable data within Federal systems. Unvalidated cryptography is viewed by NIST as providing no protection to the information or data - in effect the data would be considered unprotected plaintext. If the agency specifies that the information or data be cryptographically protected, then FIPS 140-2 is applicable. In essence, if cryptography is required, then it must be validated. Guidelines for proper key management techniques can be found in NIST Special Publication P 800-57 Part 3, Recommendation for Key Management: Application-Specific Key Management Guidance.[11]

Malicious Functionality. Some functionally-inconsistent 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 exfiltration of 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), 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 in which causes the user to unintentionally select ads that may attempt to deceive the user. These

[10] In the mathematical sense of "completeness" completeness means that no violations of the requirement will be overlooked.

[11] See NIST Special Publication P 800-57 Part 3, Recommendation for Key Management: Application-Specific Key Management Guidance http://csrc.nist.gov/publications/drafts/800-57pt3_r1/sp800_57_pt3_r1_draft.pdf

types of behaviors support phishing attacks and may not be detected by mobile antivirus of software vulnerability scanners as they are served dynamically and not available for inspection prior to the installation of the mobile app.

Excessive Permissions. Functionally inconsistent apps have permissions that are not consistent with End User License Agreements (EULAs), application permissions, application 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. 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 PII 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. It is important to note that identifying functionally inconsistent apps is a manual process. It involves a subjective decision from an evaluator that certain permissions are not appropriate and the behavior of the app insecure.

Malware. Malware is a particular challenge for evaluators because there may be a deliberate attempt to conceal it. To make matters worse, malware can be designed and tested to evade automated detection tools (such as antivirus scanners) as long as the malware author has a copy of the detection tool that he or she can use to test the malware. Because of these challenges, it is essentially impossible for a vetting process to guarantee that a piece of software is free from malicious functionality. Mobile devices do have some built-in protections against malware, for example application sandboxing and user approval in order to access subsystems such as the GPS or text messaging. But sandboxing only makes it harder for apps to interfere with one another or with the operating system; it does not prevent many of the types of malicious functionality listed above.

Vulnerabilities. Mobile apps may have residual weaknesses that make them vulnerable to attacks. Traditionally, a vulnerability is defined as a weakness that allows an attacker to reduce the system's reliability, integrity, availability, confidentiality, or functionality and is the intersection of three elements: system weakness, attacker access to the weakness, and attacker capability to exploit the weakness. To exploit vulnerability, an attacker must have at least one applicable attack tool or attack technique that can work against the system weaknesses that exploit the vulnerability. The attacker's access paths and access mechanisms to the weaknesses in a system are often referred to as the system's attack surface.

Poor Design. Mobile apps that are poorly designed may drain or misuse the device resources or have inconsistent behavior across different sets of devices. An example of such an app is one that goes into an endless loop, draining the computational resources and power rendering the device unusable. Another example is an app that has device or functional dependencies that are not disclosed to the user, altering the behavior or functional capabilities of the app across different devices. This problem has been common for some apps that have display size requirements and no clear functional specifications. As part of software testing, the functional dependencies of an application should be tested and any discrepancies or issues should be identified before the application is deployed. Commercial services are available to provide app user interface testing and usability testing on several manufacturers' devices, a variety of models and form factors.

4.2 Mobile App Tests.

Below are automated and manual tests that can be performed to evaluate how well an app meets an organization's requirements, to help them better understand the risks the apps introduce, and to decide which countermeasures may be needed to mitigate these risks.

- **User Interface:** Mobile user interface display 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 User Interface (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.
- **Malware Detection**: Although mobile malware scanning products suffer some of the same limitations of their desktop counterparts, open source and commercially-available malware detection tools can identify known and published malware signatures. Anti-virus and scanning tools can be incorporated as part of an organization’s enterprise MDM, organization’s app store, or as part of the mobile app vetting process. Note that just because an app has been scanned by an anti-virus software tool and no known malware signatures have been detected, does not mean that app does not contain vulnerabilities that can be exploited by an adversary or that the app does not perform functions that do not comply with an organization’s policies.
- **Input Validation:** Due to the short timeframes mobile apps are typically used (and relevant), it is crucial to verify data input. Inputting data, testing actions, and completing transactions confirms an app’s performance matches the stated app’s functionality from the developer. Negative test cases based on critical actions that the app is not supposed to do should also be conducted.
- **Physical Sensors:** Test any device physical sensors used by the app such as 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”). If the app initiates a telephone call or SMS, it is suggested to use a working mobile telephone number to verify the call or SMS was received.
- **Overall Functional Reliability:** Performance testing confirms that the time the app takes to display information, navigate to another page, or process a transaction is fast enough to meet user expectations. Confirm pages heavy with content, images, animations, or video load in a reasonable timeframe without crashing. Also, transactions accessing backend systems using APIs and “screen scraping” are at high risk of performance issues. If possible, it is recommended to test with different carriers, ranges of network bandwidth (LTE, 3G), and wireless internet to truly determine an app’s performance.
- **Network Events:** The network events category includes any functionality related to connections to an external network as well as internal and external content providers. All mobile apps can use network connections to retrieve content, including configuration files, from a variety of external systems. Mobile 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. Be sure to consider not only cellular and Wi-Fi network events, but also Bluetooth, NFC, and other forms of networking. Mobile apps should use the Hypertext Transfer Protocol over Secure Socket Layer (HTTPS) instead of HTTP when transferring sensitive information over the web, especially when using public Wi-Fi access points.
- **Communication with Known Disreputable Sites.** Can the app be observed communicate with sites known to harbor malvertising, spam, phishing attacks, or other malware? Does the app connect with unauthorized cloud services?
- **Protecting Data at Rest and in Transit.** Mobile apps may collect, store, and share sensitive information. Should the system owner decide that the use of cryptography is warranted, his information should be protected while at rest and in transit using FIPS-140-2 validated cryptography. Information that is transmitted without being encrypted is susceptible to eavesdropping. Encrypting data at rest can protect sensitive data on a lost or stolen mobile device. 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 web site[12].

- **Hard-coded Cryptographic Keying Material**: When properly implemented, the use of cryptography can help provide data confidentiality, authenticity, and integrity as well as other security- and privacy-enabling services. The presence of hardcoded cryptographic keying material (keys, initialization vectors, etc.) is an indication that in-app cryptography has not been properly implemented and might be used by an attacker to compromise data or network resources.
- **File 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. File scanning or access to files that are not part of an application's own directory could also be an indicator of malicious activity or bad coding practice. Files written to external storage, such as a removable SD card, may be readable and writeable by other mobile apps that may have been granted different permissions, thus placing data written to unprotected storage at risk.
- **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.
- **Privileged Commands:** Apps 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, for 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 id that the application is running under, this opens up the potential for privilege elevation attacks.
- **Libraries Loaded:** The libraries loaded category includes any third-party libraries that are loaded by the app at run time. Legitimate uses for loaded libraries might be for the use of a cryptographic library or a graphics API. Malicious apps can use library loading as a method to avoid detection. From a vetting perspective, libraries are also a concern because they introduce outside code to an application without the direct control of the developer.
- **Dynamic Behavior**: When apps execute, they exhibit numerous dynamic behaviors. Not all of these operating behaviors are solely from device user inputs. Executing apps also receive input data from information stored in the device. For example, what app data is stored, and is it stored in the sandbox or elsewhere? What external data can the app leave in the sandbox? Likewise, where does data used by the app originate from and how is it sanitized? Can a mobile app vetting system determine what data is permitted and how it affects app behavior? 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.
- **Side-channel Leakage**: Unintended channels used by an app that might make data visible to untrusted persons or apps. For example, in addition to the normal issues with concerns over what data is written to log files or cached such as user PII, passwords or failed password attempts, mobile

[12] For more information on NIST's Cryptographic Module Validation Program see http://csrc.nist.gov/groups/STM/cmvp/

devices can also capture location data that is sensitive, pictures taken by onboard cameras, both still and video, as well as the broadcast ID of the device that mobile apps may have access to and thus need to be developed and tested in a manner that protects this additional data from side-channel leakage attacks.

- **Telephony Functionality:** Telephony functionality encompass a wide variety of method calls that an app can use if given the right permissions, including making phone calls, transmitting SMS messages, and retrieving unique telephone identifier information. Most, if not all of these telephony events are sensitive and some of them provide access to personally identifiable information (PII). Many applications make use of the unique telephone identifier information in order to keep track of users instead of using a username/password scheme. Another set of sensitive calls can give an application access to the telephone number. Many legitimate applications use the telephone 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 application that makes use of the telephone call or SMS messages that is not clearly stated in the EULA or application description as intending to use them should be immediate cause for suspicion.
- **Classes Loaded:** Classes being loaded by an application can also present a security risk. Classes being loaded by an application are very similar to libraries being loaded with the exception that the classes loaded are written in Java. On the Android platform, for example, these classes are in *dex* form and contained inside of a jar file to be loaded. Dex is the file format that Java classes are compiled into for use on Android. Similar to libraries being loaded, classes being loaded can present a level of code obfuscation. Unlike library loading, class loading can be device dependent, adding one extra layer of obfuscation since some of the functionality can only manifest itself on a specific device or version of a mobile OS.
- **Inter-Application Communications:** Mobile 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.” Therefore, the Intent class and its subclasses provide a mechanism for Android apps to perform inter- and intra-application communication. An intent object can be used to start an app component in a different app or the same app that is sending the intent. Intent objects can also be used to interact and request resources from the operating system.
- **Media Events & Access to Audio/Video/Camera(s)**: Media events, similar to telephony events, are cause for immediate concern if an app does not explicitly state it intends to use them in its permissions manifest. These events are method calls relating to use of the camera (for pictures or video) and the microphone (for recording of audio). Since these events can take place completely in the background without the user’s knowledge, they are particularly suspicious for an app that does not advertise the use of them.
- **Dynamic Behavior**: Mobile app security testing identifies weaknesses that are exploitable. Review of the binary code can determine if the app is only using the appropriate APIs and where the APIs are transferring data to and from an app. Review of the running code can also determine if access control is set correctly on files, folders, communication channels, and other objects. Dynamic testing of the live app can be used to identify weaknesses in how the user interface handles input and to understand or test the “business logic” of the application (which isn’t what people traditionally think of as input).
- **Privacy and Personally Identifiable Information**: Privacy considerations, such as revealing traditional PII also include mobile-specific personal information like location data, pictures taken by onboard cameras, both still and video, as well as the broadcast ID of the device. This needs to be dealt with in the User Interface (UI) as well as in the portions of the apps that manipulate this data. For

example, a tester can verify that the app complies with privacy standards by masking characters of any sensitive data within the page display, but they should also review audit logs, when possible, for appropriate handling of this type of information. Examples of traditional types of sensitive data include financial data (e.g., credit card number), personal data (e.g., social security number) or login credentials (e.g., password). The login page should limit the number of failed authentication attempts and not provide the ability to save a password for automatic login if the app contains sensitive data. 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.

- **Excessive Resource Consumption**. Mobile apps may intentionally, or through poor programming practices, consume excessive resources such as CPU cycles, memory, external storage, networking resources, and battery power. Mobile device hardware capabilities are commensurate with those of laptops, but battery resources remain a critical resource. Mobile apps that are designed to help medical personnel or first responders need to manage and in some cases compete for limited power resources. Mobile apps that consume excessive resources should be identified so that users will know what to expect in terms of their mobile device's availability.

- **Accessibility**. Section 508 requires that when Federal agencies develop, procure, maintain, or use electronic and information technology, Federal employees with disabilities have access to and use of information and data that is comparable to the access and use by Federal employees who are not individuals with disabilities, unless an undue burden would be imposed on the agency. Section 508 also requires that individuals with disabilities, who are members of the public seeking information or services from a Federal agency, have access to and use of information and data that is comparable to that provided to the public who are not individuals with disabilities, unless an undue burden would be imposed on the agency.[13]

[13] See http://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-section-508-standards/section-508-standards for more information on Section 508 Standards for Electronic and Information Technology.

5. App Vetting Tools and Techniques

This section addresses selected considerations for mobile app vetting tools and techniques, including designing analysis processes, vetting source code versus binary code, and selecting automated tools. For basic information on security testing and assessment tools and techniques that may be helpful for performing vetting, see NIST SP 800-115, *Technical Guide to Information Security Testing and Assessment*.[14]

5.1 Designing Analysis Processes

Evaluation processes may use automated tools or may be entirely manual. Such processes may also be performed by systems that provide (semi-) automatic management of the app vetting workflow (i.e., the uploading, testing, and analysis of apps).[15] (Naturally there can also be entirely automated evaluations, but with current technology this can only be expected to yield partial results.) Furthermore, evaluation processes for security differ from those intended to assess software correctness, though correctness tests should be performed even if security is the main goal.

Software correctness testing [PY07] is a more mature field than current security analysis. The main reason is that correctness testing has been around longer, but in some ways it is also easier. First, one can argue (as a first approximation) that with appropriate test selection, if the software is unlikely to fail during testing then it is also unlikely to fail in the field [MUSA 99]. Secondly, correctness testing has traditionally been based on specifications, or at least some sort of description of what the software is supposed to do. Even if this information is sometimes hard to come by in practice, there is an expectation in the software industry that testers should have it. This is not always the case in security assessments; often the software analysts are expected to derive the requirements themselves, and automated tools are largely based on security requirements that are considered to be common across many different software artifacts. Nonetheless, correctness testing contributes to software security. For example, crashes and other unexpected behaviors are often indicative of a security flaw.

Some traditional software assurance activities may have to be omitted or adapted to the current circumstances of a mobile app vetting system It is also important to determine how many combinations of devices and operating system versions that an app may be expected to execute on can be tested based on budget, time, and resources.

5.2 Vetting Source Code versus Binary Code

A major factor in performing an app evaluation is whether the mobile app’s source code or only binary code is available. Typically, mobile apps downloaded from an app store do not provide access to source code. When source code is available, there are a variety of tools that can be used to analyze the source code or run it in an emulated environment. The goals of performing a source code review are to find vulnerabilities in the source code, and to make sure that performing risky activities in the source code is done in such a way that they are safer from a security perspective. These tasks need to be performed manually by a secure code reviewer by reading through the contents of source code files. Even with automated aids, the analysis is labor intensive. Benefits to using automated static analysis tools include

[14] See http://csrc.nist.gov/publications/nistpubs/800-115/SP800-115.pdf

[15] See NIST AppVet at http://csrc.nist.gov/projects/appvet/index.html

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 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.

When source code is not available, binary code can be vetted instead. In the context of mobile apps, the term “binary code” can refer either to 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 applications include byte-code that can be analyzed without hardware support using emulated and virtual environments.

Static analysis requires that the code be reverse engineered when source is not available, which is relatively easy for byte-code[16], 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.[17] For machine code, it is especially hard to track the flow of control across many functions and to track data flow through 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 de-compiler that tries to recover the original source code. These techniques are especially useful if the purpose of reverse engineering is to allow humans to peruse the code, since the outputs are in a form that can be understood by humans with appropriate skills. But even the best dis-assemblers make mistakes [BALA07], 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 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 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, a executing the 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.[18] Running the code on a physical device allows the tester to select the exact characteristics of the device on which the app is intended to be used on 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 of them all and the simulation may not be completely accurate.

Useful information can be gleaned by observing an app’s behavior even without knowing the purposes of individual functions. For example, the evaluator can observe how the app interacts with its external resources, recording the services it requests from the operating system and the permissions it exercises. Although many of the device capabilities used by an app are implicitly documented (since the user has to approve its access to those capabilities), this documentation tends to contain a superset of the capabilities

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

[17] Such as [FINDBUGS, SHAH05, CHEN07, ZHAO08]

[18] The Android Emulator is found in the Android SDK.

actually used. 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 evaluator might ask what data is being sent and for what purpose.

5.3 Selecting Automated Tools

Analysis tools are often characterized as being either *static* or *dynamic*. Static analysis examines the app source code and binary, including byte-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. [IEEE-2010] Organizations should consider the technical tradeoff differences between what static and dynamic tools offer and balance their usage given the organization's assurance goals.

An additional consideration is that tools may analyze either the source code of an application or the binary code. Obviously the choice of tools depends on the availability of source code and the availability of people and tools that can analyze the source code for potential issues. Evaluating source code generally can provide more extensive information on the weaknesses than testing binaries only, but testing source code requires considerably more resources. Examining source code also will not allow the vetting authority to test for and reveal any malicious action that may have taken place during the compiling process. Organizations should carefully consider this tradeoff, along with the level of risk, when determining whether to examine source code, binaries, or both.

The methods used for testing software reliability tend to be somewhat different from those used to test software for security, although reliability testing can play a role in security analysis. Accordingly, some tools are intended primarily for reliability testing and some primarily for security testing. Some cross-cutting classes of tools useful in software evaluation 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 tester 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 telephone call. Remote debuggers are a common way of examining and understanding mobile 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 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, for example, tests that will be used for many apps or executed many times (with variations) for a single app.

■ **Static Tools:**

- **Style Checkers:** These tools check program syntax to find violations of style guidelines.
- **Static Analysis Tools:** These tools generally analyze the behavior of software to find security 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.
- **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.

The SAMATE test suite provides a baseline for evaluating static analysis tools (here defined as tools that look for potential vulnerabilities by examining software behavior). 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 a MAVS engineers thought, and the SAMATE test suite is continually evolving.

Commercial automated mobile application testing tools have overlapping or complementary capabilities. For example, one tool may be based on techniques that find integer overflows[19] very reliably while another tool may be better at finding weaknesses related to command injection attacks[20]. Finding the right set of tools to evaluate a requirement can be a challenging task because of the varied capabilities of

[19] http://cwe.mitre.org/data/definitions/190.html

[20] http://cwe.mitre.org/data/definitions/77.html

diverse commercial and open source tools, and because it can be challenging to determine what the capabilities of a given tool are. 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 analysts 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.

Appendix A—App Power Consumption Testing

In this document, we opt to not categorize device battery power drainage that is a performance issue due to apps as simply a reliability issue or a security issue. There are certain vulnerabilities that can lead to denial of service attacks that drain power. However, there are also other non-malicious defects in apps that can cause power drain, thus causing device failure and reducing device reliability. Therefore, battery power drainage caused by mobile apps is both a reliability and security concern.

Power consumption testing identifies app functionality that can cause the app to consume excessive power leading to battery exhaustion. For example, an app may constantly attempt to initiate the device camera or GPS upon launching the app or during app idle time. The excessive use of a power "expensive" device resource can lead to the device being disabled as a direct result of loss of power. In practice, it is extremely difficult to infer power usage problems just by examining the app code both statically and dynamically. The reason is that neither of these processes looks for power effects; rather, they focus on the analysis of the app code as it pertains to code functionality. Another aspect that hinders power analysis is the numerous devices and configurations of operating systems. For an analysis to be accurate the specific device hardware and software configurations have to be tested. Regression techniques to extrapolate behavior within error ranges can also be applied.

To help eliminate external factors that can cause battery consumption miscalculations, the following steps can be taken before power consumption testing begins:

- Identify the hardware device that the power analysis will take place on.
- Eliminate all other applications that might be running in parallel.
- Control the testing environment in terms of availability of network, screen brightness, other active network services, etc.
- Calculate and factor in appropriate measurements the overhead of the framework that attempts to measure the power consumption.
- Measure the power under different environmental conditions (temperature, rate of measure).
- Correlate the results across at least five devices to eliminate hardware errors that might crop in.
- Create an error envelope for the measurements.

Recent research has shown that more accurate power analysis can be performed by measuring the power consumption of individual device subsystems including display, GPS, and Wi-Fi, among others. Being able to measure the power consumption of subsystems can help us model the device behavior in various operating states. Furthermore, analysis can be performed at different timescales because the measurements happen cumulatively and at real-time. For each device and operating system, we can generate consumption models that can be used to account for current power consumption and forecast future consumption within an error range. To achieve that, it is important to perform measurements that exercise individual subsystems both in isolation and in concert. The end goal is to be able to generate a power consumption model that will allow us to account for all applications and services as they operate on the device with as minimal measurement error as possible. All of the power consumption calculations should be independent of the time interval used in creating the data and should not require reduction into a fixed predefined set of states with power ratings for each.

Finally, it is recommended to use measurements of behavior for all the primary device subsystems under different conditions, which then provide an in-depth understanding of how the power is consumed on a mobile device.

Appendix B—Android Application Vulnerability Types

The purpose of this appendix is to identify and define the various types of vulnerabilities that are specific to applications running on mobile devices utilizing the Android operating system. The scope of this appendix includes application vulnerabilities for Android based mobile devices running applications 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 only focuses 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 mobile application vulnerabilities are listed below:

Table 1: Android Vulnerabilities, A Level

Type	Description	Negative Consequence
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 application without the user's consent.	An application with too many permissions may perform unintended functions outside the scope of the application's intended functionality. Additionally, the permissions are vulnerable to hijacking by another application. If too few permissions are granted, the application will not be able to perform the functions required.
Exposed Communications	Internal communications protocols are the means by which an application passes messages internally within the device, either to itself or to other applications. External communications allow information to leave the device.	Exposed internal communications allow applications 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 application.	Unintended functions could be performed outside the scope of the application's functionality.
Application Collusion	Two or more applications passing information to each other in order to increase the capabilities of one or both apps beyond their declared scope.	Collusion can allow applications to obtain data that was unintended such as a gaming application 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 application and/or subvert permissions within an application. 3. Native code (code written in languages other than Java in Android) can perform unexpected and/or malicious functionality.

Type	Description	Negative Consequence
Excessive Power Consumption	Excessive functions or unintended applications running on a device which intentionally or unintentionally drain the battery.	Shortened battery life could affect the ability to perform mission critical functions.
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 application 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
Application 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	

1113

Appendix C—iOS Application Vulnerability Types

The purpose of this appendix is to identify and define the various types of vulnerabilities that are specific to applications 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 mobile application 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 application 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 application framework and may not be extended. Unlike Android, these permissions may be modified later for individual permissions and applications.
Exposed Communication-Internal and External	Internal communications protocols allow applications to process information and communicate with other apps. External communications allow information to leave the device.	Exposed internal communications allow applications 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 application.	Unintended functions could be performed outside the scope of the application's functionality.
Application Collusion	Two or more applications passing information to each other in order to increase the capabilities of one or both apps beyond their declared scope.	Collusion can allow applications to obtain data that was unintended such as a gaming application 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 application and/or subvert permissions within an application. 3. Packed code prevents code reverse engineering and can be used to hide malware.
Excessive Power Consumption	Excessive functions or unintended applications 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 application 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
Application 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

Excessive Power Consumption	Packed Code	
	CPU Usage	
	I/O	

Appendix D—Glossary

Selected terms used in this publication are defined below.

Dynamic Analysis: Detecting the set of target weaknesses by executing an app using a set of input use cases and analyzing the app's runtime behavior.

Fault Injection Testing: Attempting to artificially trip up the 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.

Off-Nominal Testing: Exercising a mobile app to determine how it behaves under rarely expected operational scenarios.

Static Analysis: Detecting the set of target weaknesses 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." [CNSS-4009]

Target Weaknesses: A specific set of weaknesses to look for in an app.

Traditional Software Reliability Testing: Testing that determines if a mobile app does what it is supposed to do under normal, expected operational usage.

Vetting: The process of verifying that a particular app meets the requirements that all apps must follow.

Vulnerability: A weakness which allows an attacker to reduce a system's reliability, integrity, availability, confidentiality, or functionality and is the intersection of three elements: a system weakness, attacker access to the weakness, and attacker capability to exploit the weakness.

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
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DHS	Department of Homeland Security
DoD	Department of Defense
EULA	End User License Agreement
FIPS	Federal Information Processing Standard
FISMA	Federal Information Security Management Act
GPRS	General Packet Radio Services
GPS	Global Positioning System
I/O	Input/Output
IPC	Inter-Process Communications
IT	Information Technology
ITL	Information Technology Laboratory
LTE	Long-Term Evolution
MAVS	Mobile App Vetting System
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
PII	Personally Identifiable Information

RAM	Random Access Memory
SDK	Software Development Kit
SLA	Service Level Agreement
SMS	Short Message Service
SP	Special Publication
UI	User Interface
URI	Uniform Resource Identifier
USB	Universal Serial Bus
Wi-Fi	Wireless Fidelity

Appendix F—References

References for the publication are listed below.

[APP2013] Apple Energy Usage Instrument http://developer.apple.com/library/ios/#DOCUMENTATION/AnalysisTools/Reference/Instruments_User_Reference/EnergyUsageInstrument/EnergyUsageInstrument.html (Last updated: 2013-04-23)

[ASM] Java bytecode manipulation and analysis framework: http://asm.ow2.org/

[BALA07] Gogul Balakrishnan, WYSINWYX: What You See Is Not What You eXecute, Ph.D dissertation and Tech. Rep. TR-1603, Computer Sciences Department, University of Wisconsin, Madison, WI, August 2007

[BSI] Nancy R. Mead, Julia H. Allen, Sean J. Barnum, Robert J. Ellison, and Gary McGraw, "Software Security Engineering", Addison-Wesley Professional, 2008

[CHEN07] Hua Chen, Tao Zou, and Dongxia Wang. 2007. Data-flow based vulnerability analysis and java bytecode. In Proceedings of the 7th Conference on 7th WSEAS International Conference on Applied Computer Science - Volume 7 (ACS'07), Roberto Revetria, Antonella Cecchi, Maurizio Schenone, Valeri M. Mladenov, and Alexander Zemliak (Eds.), Vol. 7. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 201-207

[CNSS-4009] CNSS National Information Assurance (IA) Glossary CNSSI-4009

[CSEC] Blake Shepard, Cynthia Matuszek, C. Bruce Fraser, William Wechtenhiser, David Crabbe, Zelal Güngördü, John Jantos, Todd Hughes, Larry Lefkowitz, Michael Witbrock, Doug Lenat, and Erik Larson. 2005. A knowledge-based approach to network security: applying Cyc in the domain of network risk assessment. In Proceedings of the 17th conference on Innovative applications of artificial intelligence - Volume 3 (IAAI'05), Bruce Porter (Ed.), Vol. 3. AAAI Press 1563-1568

[DO-178B] DO-178B, Software Considerations in Airborne Systems and Equipment Certification

[DOWD06] Mark Dowd, John McDonald, and Justin Schuh, "The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities", Addison-Wesley, 2006

[FINDBUGS] Static analysis to look for bugs in Java code URL: http://findbugs.sourceforge.net/

[GAU89] Donald C. Gause and Gerald M. Weinberg, "Exploring Requirements: Quality Before Design", Dorset House, 1989

[GODE05] Godefroid, P., Klarlund, N., and Sen, K. DART: directed automated random testing. In PLDI'05: Proc. Of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (2005), ACM, pp. 213–223

[IEC-61508] IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems

[IEEE-2010] J.R. Maximoff, M.D. Trela, D.R. Kuhn, and R. Kacker, “A Method for Analyzing System State-space Coverage with a t-Wise Testing Framework”, IEEE International Systems Conference 2010, April 4-11, 2011, San Diego

[ISO 26262] ISO 26262 Road vehicles -- Functional safety

[MCGRAW05] Gary McGraw: Software Security: Building Security In, Addison-Wesley Professional, 2006

[MUSA 99] John D. Musa, Software Reliability Engineering, , McGraw-Hill, 1999

[NASA-8739] NASA Software Assurance Standard NASA-STD-8739.8

[NIST-142] D. R. Kuhn, R. N. Kacker, and Y. Lei, “Practical Combinatorial Testing”, NIST SP 800-142, 2010. http://csrc.nist.gov/groups/SNS/acts/documents/SP800-142-101006.pdf

[NUS00] Bashar Nuseibeh and Steve Easterbrook. 2000. Requirements engineering: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, 35-46

[PY07] Pezzè M. and Young, M., “Software Testing and Analysis: Process, Principles and Techniques”

[SHAH05] Shah, R.A., Vulnerability Assessment of Java Bytecode, Auburn University, 2005

[STRIDE] The STRIDE Threat Model, Microsoft Corporation, 2005

[SCHUL07] G. Gordon Schulmeyer, (Editor)Handbook of Software Quality Assurance, 4th Edition, ISBN-13: 978-1450421041,– September 30, 2007

[TRIKE] Open source threat modeling methodology and tool URL: http://www.octotrike.org

[ZHAO08] Gang Zhao, Hua Chen, and Dongxia Wang. 2008. Data-Flow Based Analysis of Java Bytecode Vulnerability. In Proceedings of the 2008 The Ninth International Conference on Web-Age Information Management (WAIM '08). IEEE Computer Society, Washington, DC, USA, 647-653