MTR160242
MITRE TECHNICAL REPORT

MITRE

Analyzing the Effectiveness of App Vetting Tools in the Enterprise

Michael Peck
Carlton Northern
August 22, 2016

Hampton, VA

Approved for Public Release; Distribution Unlimited. Case Number 16-4772

Executive Summary

Enterprises invest significant resources in mobile application (hereafter "app") vetting to determine whether apps are safe to deploy on mobile devices. App vetting seeks to identify security vulnerabilities (usually inadvertent) and malicious or privacy violating (usually deliberate) behaviors in apps. It generally involves a time- and labor-intensive effort, resulting in high costs and delays in approving apps for use. Additionally, mobile app developers often operate on a rapid development cycle, and manual vetting approaches cannot keep up with the releases of new app versions.

Various use cases for enterprise vetting of mobile apps exist, including in-house developed apps for enterprise use, in-house developed apps for public distribution, commercially developed apps for enterprise use, and commercially developed apps for personal use on enterprise devices. Each of these approaches carries different risks. This report provides guidance to US Government and commercial enterprises alike, on how to assess the feasibility of applying automated app vetting tools.

To do so, criteria has been created to evaluate the ability of the solutions to assess apps against many of the requirements in the version 1.2 of the NIAP Protection Profile for Application Software (not all of the NIAP requirements can be automated). The MITRE team additionally used NIAP's sample Mobile App Security Vetting Reciprocity Report[1], which describes useful app vetting results to report based on the NIAP PP requirements. MITRE suggests additional criteria to cover broader app vetting tool capabilities (e.g., reputation analysis), threats against the app vetting tool itself, and other common vulnerabilities or malicious behavior commonly observed in apps but not directly addressed by the NIAP PP. These criteria do not represent an exhaustive list of every potential vulnerability or malicious behavior in apps. Rather, they cover many common app issues to enable a baseline evaluation of vetting tool capabilities. Enterprises may add criteria based on their specific needs and risks.

On the basis of the assessment criteria, the MITRE team developed or obtained several vulnerable and potentially malicious apps to use in assessing the ability of app vetting solutions. The results from testing these apps provide a basic, high-level baseline for assessing app vetting solution capabilities to address the criteria. The apps developed/selected do not exhibit every potential behavior that violates the criteria and thus the results do not ensure the ability of app vetting solutions to provide full code coverage. For example, vulnerable or malicious code could be hidden behind a login prompt or other user interface element or behind a deliberately inserted time delay before execution, or could make use of emulator detection to evade analysis. The tools may or may not detect these and other kinds of evasion techniques.

MITRE conducted a market analysis of mobile app vetting tools and found 30 such commercial or open source offerings. The enclosed *App Vetting Tools Market* Analysis spreadsheet contains the findings of the market analysis. Given the limited resources of time and funding, MITRE evaluated only those tools most likely to satisfy the largest number of identified criteria. As such, the evaluation gave preference to tools that claimed conformance with the NIAP PP; the team also looked to Gartner studies[2][3] to select top performers.

[1] https://github.com/commoncriteria/application/wiki/Schema
[2] Gartner's Application Security Testing Magic Quadrant 2015
[3] Gartner's Critical Capabilities for Application Security Testing 2015 – Mobile App Testing

The tools evaluated were Android Lint (Included in Android Studio and Android SDK) and 8 other commercial products that have been anonymized due to non-disclosure agreements between MITRE and their respective companies. The products are identified by Product 1 – 8.

Overall, MITRE found that the best solutions generally employed a combination of static and dynamic analysis techniques. Static analysis techniques provide insights into the properties of the app and can detect many vulnerabilities, while dynamic analysis techniques reveal app behaviors that only occur at runtime.

Solution capabilities vary widely. Some solutions focus on traditional Java or other language coding issues and have limited coverage of weaknesses specific to high-risk Android and iOS mobile apps. Other solutions have a strong focus on specific weaknesses that commonly occur in Android and iOS apps. Therefore, enterprises should carefully assess capabilities and choose the best solution or solutions to meet their particular needs.

Some solutions can identify both vulnerabilities and potentially malicious or privacy-violating behaviors. However, in the solutions examined MITRE found that it would be trivial for a malicious app to identify the presence of an analysis environment versus a typical mobile device being used by a real user, and to adapt its behavior accordingly to evade detection. Even the solutions that use real mobile devices rather than emulators contain obvious indicators of an analysis environment, such as the presence of the Xposed framework (Android) or the Cydia app store (iOS) in the list of installed apps. Identifying vulnerabilities and identifying potentially malicious behaviors represent different use cases, and robustly detecting malicious applications is very difficult.

Acknowledgments

The authors would like to acknowledge the contributions to this report by Jay Vora, Gavin Black, Sarah Ford, Dustin Hooven, Michael Schoenfeld and Rushi Purohit.

This technical data deliverable was developed using contract funds under Basic Contract No. W15P7T-13-C-A802.

Table of Contents

1 Introduction 1
1.1 Background 1
1.2 Document Structure 1
2 Mobile App Use Cases and Associated Risks 3
3 Evaluation Criteria for App Vetting Solutions 7
3.1 Evaluation Criteria 7
3.2 Test Cases 12
4 Vulnerable and Potentially Malicious Test Apps 19
4.1 Android Apps 19
4.1.1 UploadDataApp 19
4.1.2 Custom-class-loader 21
4.1.3 Device Administrator Sample App 22
4.1.4 Android Vulnerability Test Suite 22
4.1.5 Subterfuge App 22
4.2 iOS App 23
4.2.1 Acme Airlines 23
5 Tool Evaluation 25
5.1 Vetting Tools Examined 25
5.2 Findings 25
6 Future Work 31
Appendix A Android Platform Mitigations A-1
A.1 Access to Hardware Resources and Sensitive Information Repositories (NIAP App PP FDP_DEC_EXT.1.1 and FDP_DEC_EXT.1.2) A-1
A.2 Sensitive Application Data / File Permissions (FDP_DAR_EXT.1.1 and FMT_CFG_EXT.1.2) A-3
A.3 Network Communication (FDP_NET_EXT.1.1) / Protecting Data in Transit A-4
Appendix B NIAP Protection Profile Recommendations B-1
Appendix C Tool Assessments C-1
C.1 Android Lint C-1
Appendix D Acronyms D-1

List of Figures

Figure A-1. Android's Runtime Permissions A-2
Figure A-2. Old Android Permissions A-3
Figure C-1. Android Lint Screenshot..... C-1

List of Tables

Table 1. Primary App Vetting Focus Areas 5
Table 2. Mobile App Vetting Test Cases 12
Table 3. Android Test Results 27
Table 4. iOS Test Results 29

1 Introduction

1.1 Background

Enterprises invest significant resources in mobile application (hereafter "app") vetting to determine whether apps are safe to deploy on mobile devices. App vetting seeks to identify security vulnerabilities (usually inadvertent) and malicious or privacy violating (usually deliberate) behaviors in apps. It generally involves a time- and labor-intensive effort, resulting in high costs and delays in approving apps for use. Additionally, mobile app developers often operate on a rapid development cycle, and manual vetting approaches cannot keep up with the releases of new app versions.

Various use cases for enterprise vetting of mobile apps exist, including in-house developed apps for enterprise use, in-house developed apps for public distribution, commercially developed apps for enterprise use, and commercially developed apps for personal use on enterprise devices. Each of these approaches carries different risks. This report provides guidance to US Government and non-government enterprises alike, on how to assess the feasibility of applying automated app vetting tools.

Numerous vendors now provide automated app vetting tools (sometimes also known as app threat intelligence or threat protection services) that run static and/or dynamic analysis tests on apps to detect security vulnerabilities, maliciousness, or privacy-violating behaviors. These tools may be provided as cloud-based services or as on-premises solutions. Many of the tools regularly crawl commercial app stores, automatically analyzing new app versions using the vendors' evolving knowledge of mobile threats and making the analysis results available. Some can perform reputational analysis of apps and their developers based on intelligence information gathered from sources such as app stores and participating mobile devices. They may also provide the ability to directly submit in-house apps for analysis that may not be present on the mainstream app stores. Leveraging these commercial offerings enables enterprises to streamline app vetting, decreasing the cost and time associated with analysis while potentially improving security by staying up-to-date with emerging threats and app versions.

1.2 Document Structure

Section 2 of this report contains background on mobile app use cases and associated risks addressed during app vetting.

Section 3 summarizes repeatable criteria suitable for assessing app vetting tools. The criteria are primarily based on the National Information Assurance Partnership's (NIAP's) Protection Profile (PP) for Application Software v1.2,[4] which was adopted by the Federal CIO (Chief Information Officer) Council as the source of app vetting criteria for government use. The PP contains requirements intended to address both vulnerabilities and malicious or privacy violating behavior. This section also details Android-specific and iOS-specific test cases to apply to app vetting tools.

Section 4 describes sample Android and iOS apps that the MITRE research team developed or obtained that exhibit various vulnerabilities and suspicious behaviors aligned with the criteria

[4] https://www.niap-ccevs.org/pp/PP_APP_v1.2/

described in section 3. These sample apps can be used to assess app vetting tools against the criteria.

Section 5 lists the anonymized tools that the team evaluated and the selection criteria that was used to select them, as well as high-level, aggregated conclusions from applying MITRE's test cases to them.

Section 6 poses future work possibilities.

2 Mobile App Use Cases and Associated Risks

As described above, mobile app vetting may include searches both for potentially exploitable vulnerabilities and for potentially malicious or privacy-violating behaviors. Each enterprise must determine its own acceptable level of risk when deciding on the necessary scope of security analysis. However, MITRE recommends taking into account both the organization that developed the app (in-house vs. external) as well as the planned use of the app (whether it will be used for enterprise purposes or not). In all cases, enterprises should also take into account the security features provided by the mobile platform (operating system and other underlying on-device technologies as well as broader ecosystem capabilities). Many mobile app analysis tools can quickly perform a reputational analysis of the app and its developer, which in some environments may be sufficient to justify approving an app for use.

Apps developed in-house are less likely to contain intentionally malicious or privacy-violating functionality than external apps. Vetting of these apps can therefore focus primarily on searching for security vulnerabilities. However, some level of risk of malicious or privacy violating behavior may still exist: for example, third-party software libraries included in the app may include privacy-violating behaviors to enable targeted advertising that are not known to the app developer. Similarly, if the enterprise outsourced all or part of development, it may not be aware of the full behavior of the app, yet might still be held responsible for the app's behavior.

It is important to note that no known analysis tool can perform an exhaustive search for all possible vulnerabilities, maliciousness, or privacy-violating behavior. Enterprises should follow secure software development practices when developing apps in-house and ensure they understand and mitigate any potential implications of using third-party libraries or outsourced software development.

Personal apps are not intended to process enterprise data. Vetting of these apps can primarily focus on searching for malicious or privacy-violating functionality, with a search for security vulnerabilities being less critical to the enterprise, because mobile device app sandboxes generally isolate the impact that exploitation of a single app would have on other apps on the device. However, some risk still exists that an attacker could use a vulnerable app as a vector to exploit the mobile device itself and gain access to enterprise data. Additionally, individual mobile device users would certainly be interested in the potential impact on their personal data of any vulnerabilities in personal apps.

Table 1 summarizes the recommended primary focus areas for app vetting based on application developer and planned use of app.

Table 1. Primary App Vetting Focus Areas

	Malicious Functionality	Security Vulnerabilities
In-house enterprise use app	Less critical, but may still be necessary	Primary focus
In-house developed app for public distribution	Less critical, but may still be necessary	Primary focus
Commercial enterprise use app	Both should be examined	Both should be examined
Commercial personal use app	Primary focus	Less critical, but may still be necessary

Enterprises should take the properties provided by modern mobile platforms into account when determining the required scope of app vetting. The NIAP Protection Profile for Application Software takes many of these properties into account in its operating system-specific tests for each requirement. Mobile operating systems contain built-in security features designed to provide protection from malicious behaviors, decrease the likelihood of vulnerabilities, and decrease the impact of exploitation of vulnerabilities. Additionally, the broader mobile ecosystem provides protections as well.

Typically, the operating system prohibits mobile apps from accessing data stored by other apps and from interfering with the behavior of other apps. Apps must request permission to access sensitive information repositories (e.g., contact list) and sensitive device hardware capabilities (e.g., microphone, camera, Global Positioning System [GPS]). The operating system also limits apps' ability to access other underlying device resources and services. For example, on Android, all apps must include a manifest file (AndroidManifest.xml) that defines the app's permissions and other important properties, and the operating system enforces the contents of the manifest file.

In an effort to reduce the presence of exploitable vulnerabilities, mobile operating systems provide safe default behaviors in many cases. For example, https or other Transport Layer Security (TLS)-secured connections perform proper certificate validation by default, although they are sometimes inadvertently overridden by individual apps to perform insecure behaviors. The Network Security Configuration feature introduced in Android 7 (Nougat) and the App Transport Security feature introduced in iOS 9 will likely help app developers avoid many common network security mistakes.

Device or operating system-level features such as Android for Work, Samsung KNOX Workspace, and Apple iOS-managed apps can be used to separate apps processing enterprise data from those that do not. These features to some extent defend enterprise apps against vulnerabilities or malicious activities within apps that do not process enterprise data.

The broader mobile ecosystem also provides additional protections. For example, both Google and Apple screen apps submitted to their app stores for both vulnerabilities and harmful behaviors, and have mechanisms in place to discourage installation of apps from other sources. Google gives details of its app security analysis process on pages 14–24 of its Android Security

2015 Year in Review report.[5] Google's report also contains an overview of the Android mobile ecosystem security protections in general.

Appendix A provides additional related information on mitigations provided by the mobile platforms. Malicious apps can certainly attempt to exploit vulnerabilities in the operating system in order to escalate privileges and bypass these protections, but doing so involves significant extra effort, increased risk of detection, and decreased probability of success. The likelihood of success varies between mobile platforms.

On the other hand, the mobile environment can exacerbate the impact of some app weaknesses. For example, mobile devices are commonly used on unprotected public Wi-Fi networks, where an attacker can easily intercept or manipulate network communication. Therefore, enterprises must generally treat data-in-transit issues such as use of plaintext network communication (for instance, using http instead of https) or improperly configured network encryption (for instance, disabling certificate validation) as critical issues to address.

[5] http://static.googleusercontent.com/media/source.android.com/en//security/reports/Google_Android_Security_2015_Report_Final.pdf

3 Evaluation Criteria for App Vetting Solutions

3.1 Evaluation Criteria

This section provides high-level criteria for enterprises to use in evaluating app vetting solutions. This report presents criteria to evaluate the ability of the solutions to assess apps against many of the requirements in the version 1.2 of the NIAP Protection Profile for Application Software (not all of the NIAP requirements can be automated). The MITRE team additionally used NIAP's sample Mobile App Security Vetting Reciprocity Report[6], which describes useful app vetting results to report based on the NIAP PP requirements. MITRE suggests additional criteria to cover broader app vetting tool capabilities (e.g., reputation analysis), threats against the app vetting tool itself, and other common vulnerabilities or malicious behavior commonly observed in apps but not directly addressed by the NIAP PP. These criteria do not represent an exhaustive list of every potential vulnerability or malicious behavior in apps. Rather, they cover many common app issues to enable a baseline evaluation of vetting tool capabilities. Enterprises may add criteria based on their specific needs and risks. The following list presents the recommended criteria for evaluating app vetting solutions and identifies the relevant NIAP PP requirement if one exists:

1. Types of applications supported by the app vetting solution:

- A. Supported platforms: e.g., Android, iOS, Windows
- B. App source code required or not required
- C. What types of application code can be assessed? For example, on Android, can only the Java code be assessed, or can native code also be assessed? Are cross-platform app development frameworks supported (for example, Apache Cordova)?

2. Ability to assess general risks associated with an app [This item can be skipped for in-house developed apps]:

- A. Assess ability to assess the reputation of the app and its developer; for example:
  - Does the same app exist in mainstream app stores? How popular is it (determined by number of downloads and ratings)?
  - How many apps from the same developer exist in mainstream app stores? How popular are they? Have security issues been found in other apps from the same developer?
- B. Are there indications that the app is repackaged/counterfeit? For example:
  - Does another app with the same name but from a different developer exist in mainstream app stores and/or is installed on a large number of mobile devices?

3. Ability to detect potentially exploitable security vulnerabilities:

- A. Assess ability to identify failure to invoke an appropriate random number generator where needed.

[6] https://github.com/commoncriteria/application/wiki/Schema

- NIAP FCS_RBG_EXT.1.1

- B. Assess ability to identify insecurely storing private keys, passwords, or related secret values.
  - NIAP FCS_STO_EXT.1.1
- C. Assess ability to report data stored by the app, including whether the data is stored securely (e.g., in an appropriate storage location and with appropriate file permissions).
  - NIAP FDP_DAR_EXT.1.1 and FMT_CFG_EXT.1.2
- D. Assess ability to report whether network communications use secure protocols (e.g., HTTPS vs. HTTP) and any related security issues such as improper HTTPS/TLS certificate validation or hostname checking.
  - NIAP FTP_DIT_EXT.1.1, FCS_TLSC_EXT.1.1, FCS_TLSC_EXT.1.2, FCS_TLSC_EXT.1.3, FIA_X509_EXT.1.1
- E. Assess ability to identify default credentials found within the app.
  - NIAP FMT_CFG_EXT.1.1
- F. Assess ability to identify mapping of memory at explicit locations.
  - NIAP FPT_AEX_EXT.1.1
- G. Assess ability to identify mapping of memory as both writable and executable.
  - NIAP FPT_AEX_EXT.1.2
- H. Assess ability to determine whether the app successfully runs on the latest version of the operating system (and hence is likely compatible with its security architecture).
  - NIAP FPT_AEX_EXT.1.3
- J. Assess ability to determine whether the app places executable code in locations where the code can be modified.
  - NIAP FPT_AEX_EXT.1.4
- K. Assess ability to identify code compiled without stack-based buffer overflow protection enabled.
  - NIAP FPT_AEX_EXT.1.5
- L. Assess ability to identify third-party libraries included in the app.
  - NIAP FPT_LIB_EXT.1.1
- M. Assess ability to identify other common cryptographic implementation issues.
- N. Assess ability to identify inter-process communication issues.
  - [Android only] Assess ability to identify app components (activities, services, broadcast receivers, content providers) that are exported and could expose potential attack surface to other apps. Also assess ability to detect specific, common interprocess communication issues such as vulnerable broadcast receivers for

protected-broadcast action strings that do not properly check the received action string (see CWE-925 for more details).

- No directly applicable NIAP requirement.
- The MITRE team has regularly identified these issues in Android applications as part of its own research and code review activities.

4. Ability to detect potentially malicious or privacy-violating behaviors:

- A. Assess ability to report hardware resources accessed by the app. Report permissions requested by the app and/or actual operations performed.
  - NIAP FDP_DEC_EXT.1.1
- B. Assess ability to report sensitive information repositories accessed by the app. Report permissions requested by the app and/or actual operations performed.
  - NIAP FDP_DEC_EXT.1.2
- C. Assess ability to report all network communication performed by the app.
  - NIAP FDP_NET_EXT.1.1 (was FDP_DEC_EXT.1.4)
- D. Assess ability to determine whether the application attempts to update executable code after installation.
  - NIAP FPT_TUD_EXT.1.4
- E. Assess ability to ensure the app only uses supported platform application programming interfaces (APIs).
  - NIAP FPT_API_EXT.1.1
- F. Assess ability to determine whether the application code has been obfuscated or otherwise deliberately implemented in a way that makes security analysis more difficult.
  - NIAP AVA_VAN.1.1C
  - NOTE: Some app vetting tools identify the lack of obfuscation as a security issue, e.g., they encourage the use of obfuscation—the opposite of what the NIAP App PP calls for. The Open Web Application Security Project (OWASP) Mobile Security Project has also called for the use of obfuscation. The MITRE team is not yet sure how to reconcile this contradiction. In some cases, it may be possible to use an un-obfuscated debug version of the app for vetting purposes and a separate obfuscated version for the actual production-released app.
- G. Assess ability to identify known malicious code (e.g., operating system exploits) within the application.
  - NIAP AVA_VAN.1.2E
- H. [Android only] Assess ability to identify attempts by the application to obtain device administrator access.
  - No applicable NIAP requirement

- MITRE added this requirement because examples exist of malicious apps that request and then abuse device administrator access. For example, ransomware apps can abuse device administrator access to reset the device's screen lock password and lock the user out of his or her own device. One example is the Xbot family of malicious apps reported by Palo Alto Networks.[7]

- I. [iOS Only] Assess ability to identify Uniform Resource Locator (URL) scheme hijacking issues, where the app registers URL schemes that belong to other apps in order to hijack communications intended for that other app.
  - No directly applicable NIAP requirement

5. Security of the app vetting tool itself:

- A. Ability to resist attempts by malicious apps to determine that they are running in an analysis environment.
- B. [Cloud-based vetting solutions used by multiple tenants] Ability to refresh the environment after each app is analyzed so that sensitive data is not exposed to other apps under analysis.
- C. Ability to resist persistent exploitation of the analysis environment by a malicious app. (MITRE did not actually test any tools against this requirement, but believes it should be taken into consideration for future work.)

6. Reporting capabilities:

- A. Ability to list output formats supported, e.g., JSON, XML, other machine-consumable data format, XLS, PDF, etc.
- B. Ability to provide evidence (e.g., network packet captures, system call traces, screenshots) that analysts can use to clarify or confirm reported information about apps.
- C. Ability to integrate with Enterprise Mobile Management/Mobile Device Management (EMM/MDM) systems to:
  - Automatically analyze apps installed on the enterprise's mobile devices using the EMM/MDM system's knowledge of device app inventory.
  - Respond when apps with issues are identified (e.g., alert an administrator, uninstall the offending app, limit device access to enterprise resources until the issue is resolved).

7. Other iOS tests (Note: these are not specific to iOS only, but due to time constraints were not implemented in the Android apps):

- A. Assess ability to find instances of unsanitized user input that could be used for a host of attacks, to include SQL injection and Cross-Site Scripting.
- B. Assess ability of the tool to detect time bomb attacks; e.g., execute code only 7 days after installation.

[7] http://researchcenter.paloaltonetworks.com/2016/02/new-android-trojan-xbot-phishes-credit-cards-and-bank-accounts-encrypts-devices-for-ransom/

Note: MITRE did not include the following mandatory NIAP Protection Profile for Application Software Security Functional Requirements in the above criteria:

- FPR_ANO_EXT.1.1: Not automatable because typically app vetting tools have no objective way to automatically distinguish personally identifiable (PII) from non-PII.
- FMT_MEC_EXT.1.1: Not automatable because typically app vetting tools have no automated way to determine what constitutes a configuration option being stored or set through an improper mechanism.
- FMT_CFG_EXT.1.1: Not automatable.
- FMT_SMF.1.1: Not automatable.
- FPT_TUD_EXT.1.1: Not automatable.
- FPT_TUD_EXT.1.2: No test to perform on Android and iOS, as apps are always in the platform-supported package manager format.
- FPT_TUD_EXT.1.3: Not automatable.
- FPT_TUD_EXT.1.5: Not automatable.
- FPT_TUD_EXT.1.6: No test to perform on Android and iOS, as apps are always signed.

The NIAP PP contains a number of optional and selection-based requirements. The MITRE team generally did not include those in the criteria listed above.

3.2 Test Cases

Table 2 lists the test cases for Android and iOS.

Table 2. Mobile App Vetting Test Cases

Assessment Criteria	Test Case Goal	Android Test Cases	iOS Test Cases
2A. Ability to assess the reputation of the app and its developer. 2B. Indications that the app is counterfeit/ repackaged?	Assess ability to determine the reputation of the app and its developer. Assess ability to determine whether the app is counterfeit/ repackaged.	Test a well-known commercial app: Did the tool provide an appropriate reputation result (based on information such as the presence of the same app in a mainstream app store, its popularity, its reviews, etc.)?	Test a well-known commercial app: Did the tool provide an appropriate reputation result (based on information such as the presence of the same app in a mainstream app store, its popularity, its reviews, etc.)?
		Repackage a well-known commercial app: Did the tool report it as a counterfeit/repackaged app?	Repackage a well-known commercial app: Did the tool report it as a counterfeit/repackaged app?
3A / NIAP FCS_RBG_EXT.1.1	Assess ability to identify failure to invoke an appropriate random number generator.	UploadDataApp: Did the tool detect the use of a static IV for AES-CBC encryption rather than the appropriate practice of generating a random IV?	AcmeAirlines: Did the tool detect the use of a static IV for AES-CBC encryption rather than the appropriate practice of generating a random IV?
3B / NIAP FCS_STO_EXT.1.1	Assess ability to identify passwords or other sensitive data improperly stored on the device.	UploadDataApp: Did the tool detect the password value written to the app's internal storage directory? Did the tool detect the password value written to the device's external storage directory (/sdcard)? This test would likely require that the tool provide a mechanism to input tainted sensitive data values to then scan for.	AcmeAirlines: Did the tool detect the password value written to the app's internal storage directory? This test would likely require that the tool provide a mechanism to input tainted sensitive data values to then scan for.
3C / NIAP FDP_DAR_EXT.1.1 and FMT_CFG_EXT.1.2	Assess ability to report data stored by the app, including whether it is stored securely (e.g., in an	Did the tool report the files written to internal storage with deliberately insecure world-readable and world-writable file permissions?	Did the tool report files written to internal storage with *NSFileProtectionNone* or "No Protection"?

	appropriate directory and with appropriate file permissions).	Did the tool report the files written to external storage where they could be potentially accessed by other apps?	N/A
3D / NIAP FDP_DIT_EXT.1.1, FCS_TLSC_EXT.1.1, FCS_TLSC_EXT.1.2, FCS_TLSC_EXT.1.3, FIA_X509_EXT.1.1	Assess ability to identify insecure network communication, including the use of plaintext protocols, improper certificate validation, or improper hostname checking.	Did the tool report the app's network communication, including the destination address and port?	Did the tool report the app's network communication, including the destination address and port?
		Did the tool report the transmission of sensitive data in cleartext (the password used for HTTP Basic Authentication to the server)?	Did the tool report the transmission of sensitive data in cleartext (the password used for HTTP Basic Authentication to the server)?
		Did the tool report that server certificate checking and hostname verification was disabled, making the app's network communication susceptible to man-in-the-middle attacks?	Did the tool report that server certificate checking and hostname verification was disabled, making the app's network communication susceptible to man-in-the-middle attacks?
3E / NIAP FMT_CFG_EXT.1.1	Assess ability to identify default credentials embedded in the application code.	UploadDataApp: Did the tool detect either the static AES key and IV embedded in the app and used for encryption, or the username and password embedded in the app and used for HTTP authentication?	Did the tool detect either the static AES key and IV embedded in the app and used for encryption, or the username and password embedded in the app and used for HTTP authentication?
3F / NIAP FPT_AEX_EXT.1.1	Assess ability to identify mapping of memory at explicit locations.	Assess custom-class-loader. Did the tool detect that the app maps memory at an explicit address?	N/A
3G / NIAP FPT_AEX_EXT.1.2	Assess ability to identify mapping of memory as both writable and executable.	Did the tool detect that the app maps memory with both write and execute permissions?	N/A

3H / NIAP FPT_AEX_EXT.1.3	Assess ability to determine whether the app properly leverages OS anti-exploitation capabilities where appropriate.	Assess any app: Was the tool able to identify whether the app could successfully start up on the latest released version of Android or is the tool able to successfully scan code written for the latest version of Android?	Assess any app: Was the tool able to identify whether the app could successfully start up on the latest released version of iOS? Note: iOS does not face the compatibility issues that Android does. MITRE has developed AcmeAirlines for the most current version of iOS so the test should be sufficient.
3J / NIAP FPT_AEX_EXT.1.4	Assess ability to determine whether the app places executable code in locations where the code can be modified.	Combine with test for 4D / NIAP FPT_TUD_EXT.1.4.	N/A
3K / NIAP FPT_AEX_EXT.1.5	Assess ability to identify code compiled without stack-based buffer overflow protection enabled.	Custom-class-loader: Determine whether the lack of stack protection buffer overflow was identified.	N/A
3L / NIAP FPT_LIB_EXT.1.1	Assess ability to identify third-party libraries included in the app.	Subterfuge app (APK available): Did the tool identify the presence of OpenSSL? Did it identify the version number (1.0.1o)? Did it identify relevant security issues (e.g., OpenSSL 1.0.1o is susceptible to CVE-2015-1793)?	Acme Airlines: Did the tool show the use of iOS Libraries, Frameworks and Pods used? Did it report the use of JSPatch pod?
3M / Ability to identify other crypto issues not addressed above	Assess ability to identify other crypto issues not addressed above.	Did the tool detect the use of AES in CBC mode without an accompanying authentication mechanism (e.g., MAC computation)?	Did the tool detect the use of AES in CBC mode without an accompanying authentication mechanism (e.g., MAC computation)?

3N Ability to identify inter-app communica-tion security issues	Assess ability to identify app components (activities, services, broadcast receivers, content providers) that are exported and could expose potential attack surface to other apps. Assess ability to identify vulnerable broadcast receivers for protected-broadcast action strings that do not properly check the received action string (CWE-925).	UploadDataApp: Did the tool detect that SendIntentService, LocationService, RecordIntentService, and InjectSMSService are exported and hence can be directly invoked by other apps on the device? UploadDataApp: Did the tool detect that SMSReceiver does not check the sender's permission?	Acme Airlines: Did the tool show the user which URI schemes the app had registered? This will aid the evaluator in determining if a URI scheme is trying to be hijacked.
		UploadDataApp: Did the tool detect that BootReceiver does not check that the received intent's action string matches the expected value of "android.intent.action.BOOT_COMPLETED"?	N/A
4A / NIAP FDP_DEC_EXT.1.1 4B / NIAP FDP_DEC_EXT.1.2	Assess ability to report hardware resources accessed by the app. Report permissions requested by the app and/or actual operations performed. Assess ability to report sensitive information repositories accessed by the app (e.g., calendar, call logs, contact list/address book, system logs, photos). Report permissions requested by the app and/or actual operations performed.	Did the tool report the permissions requested by the app to hardware resources and sensitive information repositories?	Did the tool report the permissions requested by the app to hardware resources and sensitive information repositories?
		Did the tool detect: - Audio recorded using microphone - GPS location gathered	Did the tool detect: - Audio recorded using microphone - GPS location gathered
		Did the tool detect: - Contact list access - Call log access - IMEI access - Phone number access - Accessing names of all files in external storage - Accessing incoming SMS messages - Installed apps list - Sending all information described above over the network	Did the tool detect: -Contact list access -Installed apps list -Sending all information described above over the network
4C / NIAP FDP_NET_EXT.1.1	Assess ability to report all network communication performed by the app.	Combine with test 3D – no need to report results in this row.	Combine with test 3D – no need to report results in this row.

4D / NIAP FPT_TUD_EXT.1.4	Assess ability to determine whether the application attempts to update executable code after installation.	custom-class-loader: Did the tool detect that the app downloads new Dalvik and native code from a remote server and executes it (secondary_dex.jar and libhello-jni.so)?	Did the tool report usage of JSPatch?
4E / NIAP FPT_API_EXT.1.1	Assess ability to ensure the application only uses supported platform APIs.	UploadDataApp: Did the tool detect the call (via reflection) to com.android.internal.telephony.GsmAlphabet.stringToGsm7BitPacked, which is an unsupported platform API?	Did the tool report the usage of a private API as demonstrated by using the *allApplications* selector from the LSApplicationWorkspace class?
4F / NIAP AVA_VAN.1.1C	Assess ability to determine whether the application code has been obfuscated or otherwise deliberately implemented in a way that makes security analysis more difficult.	Run one of the above apps through ProGuard or other obfuscation tool and determine if the app vetting tool identifies the use of obfuscation.	Run one of the above apps through PPiOS-Rename or other obfuscation tool and determine if the app vetting tool identifies the use of obfuscation.
4G / NIAP AVA_VAN.1.2E	Assess ability to identify known malicious code (e.g., OS exploits) within the application.	Did the tool identify NowSecure VTS app's tests for known device vulnerabilities as potential malicious code?	N/A
4H / Ability to identify attempts by the app to obtain device administrator access	Assess ability to identify attempts by the application to obtain device administrator access.	Device Admin sample app: Did the tool report the app's use of device administrator access?	N/A
5A / Ability to resist attempts by malicious apps to determine that they are running in an analysis environment	Assess ability to resist attempts by malicious apps to determine that they're running in an analysis environment.	UploadDataApp: Examine the data collected for potential indicators of an analysis environment (e.g., OS version numbers, names of installed apps, device serial number, IMEI, phone number, etc.)	AcmeAirlines: Examine the data collected by for potential indicators of an analysis environment (e.g., OS version numbers, names of installed apps, , etc.) Did the tool report jailbreak detection of the app? Optional: making sure the ptrace and syscall functions are not commented out, see if the app crashes or the analysis does not proceed as expected.

5B / Ability to refresh environment after each app is analyzed so that sensitive data is not exposed to other apps under analysis	Assess ability of app analysis service (particularly cloud-based services) to protect information about other customers and apps.	UploadDataApp: Examine the data collected for indications of other apps that were previously analyzed. Data to examine includes the list of other currently installed apps as well as the contents of the external storage directory (typically /sdcard).	AcmeAirlines: Examine the data collected for indications of other apps that were previously analyzed. Data to examine includes the list of other currently installed apps as any other data left on the device (contacts, pictures, etc.).
6A / Ability to determine the output formats available for reporting		Examine product and list the available output formats.	Examine product and list the available output formats.
6B / Ability to provide evidence (e.g., network packet captures, system call traces, screenshots) that analysts can use to clarify or confirm reported information about apps	Ability to provide evidence (e.g., network packet captures, system call traces, screenshots) that can be used by analysts to gain context around reported issues, e.g., to confirm the accuracy reported results, or to determine if the reported results present true risks given the app's intended purpose.	Examine the output of the app vetting tool to determine if this information is provided.	Examine the output of the app vetting tool to determine if this information is provided.
6C / Ability to determine enterprise integration capabilities.	Determine ability to integrate with other enterprise security systems - in particular, EMM/MDM systems - to gather relevant data and to enforce security policies.	Examine product and detail what EMM/MDM systems it is capable of integrating with (or available interfaces/APIs that could be used by an enterprise to implement such integration) and the level of integration (e.g., ability to gather app inventory from the EMM/MDM, ability to enforce compliance with enterprise app policies and respond to application-based threats through integration with the EMM/MDM).	Examine product and detail what EMM/MDM systems it is capable of integrating with (or available interfaces/APIs that could be used by an enterprise to implement such integration) and the level of integration (e.g., ability to gather app inventory from the EMM/MDM, ability to enforce compliance with enterprise app policies and respond to application-based threats through integration with the EMM/MDM).

7A / Ability to unsanitized input (iOS)		N/A	Demonstrating unsanitized user input that could be used for SQL injection or XSS.
7B / Ability to code coverage (iOS)		N/A	Load calendar entries 7 days after app install to see if the tool can find timed exploits. (complete code coverage)

4 Vulnerable and Potentially Malicious Test Apps

On the basis of the assessment criteria described above, the MITRE team developed or obtained several vulnerable and potentially malicious apps to use in assessing the ability of app vetting solutions. The subsections below describe these apps with reference to requirements in section 3, and, when applicable, to corresponding NIAP Application PP requirements.

The results from testing these apps provide a basic, high-level baseline for assessing app vetting solution capabilities to address the criteria. The apps selected do not exhibit every potential behavior that violates the criteria and thus the results do not ensure the ability of app vetting solutions to provide full code coverage. For example, vulnerable or malicious code could be hidden behind a login prompt or other user interface element or behind a deliberately inserted time delay before execution, or could make use of emulator detection to evade analysis. The tools may or may not detect these and other kinds of evasion techniques.

Some of these applications have been approved for public release at MITRE or were found on the open web. A collection of these applications are being maintained publicly here: https://github.com/mitre/vulnerable-mobile-apps. The UploadDataApp and AcmeAirlines app are currently in the public release process and will be uploaded to this location once approved.

4.1 Android Apps

4.1.1 UploadDataApp

This app demonstrates the following vulnerabilities:

- Access to device hardware resources (4A / NIAP FDP_DEC_EXT.1.1) and to sensitive information repositories on the device (4B / NIAP FDP_DEC_EXT.1.2)
- Insecure writing of sensitive app data to device storage (3C / NIAP FDP_DAR_EXT.1.1 and FMT_CFG_EXT.1.2), and insecure network communication (3D / NIAP FDP_DIT_EXT.1.1, FIA_X509_EXT.1.1, FCS_TLSC_EXT.1.2, FCS_TLSC_EXT.1.3, 4C / FDP_NET_EXT.1.1)
- Inclusion of default credentials (3E / NIAP FMT_CFG_EXT.1.1) and insecure storage of credentials (3B / NIAP FCS_STO_EXT.1.1
- Failure to invoke an appropriate random number generator where needed (3A / NIAP FCS_RBG_EXT.1.1) and other inappropriate cryptographic practices (3M)
- Use of an unsupported platform API (4E / NIAP FPT_API_EXT.1.1).

At app start time, the app demonstrates it has established access to device hardware resources by attempting to activate the device microphone for 5 seconds and sending the recorded audio to a remote server. The app additionally attempts to gather the device's physical location (e.g., through GPS) and send the location data to a remote server. The app always attempts to obtain Short Message Service (SMS) messages received by the device and send them to a remote server.

At start time, the app uses Android APIs to attempt to gather information from the device's sensitive information repositories and send the gathered information to a remote server via HTTPS (it can also be configured to use HTTP). This information includes:

- Whether the Android Debug Bridge (USB debugging) is on or off
- Whether installation of non-Google Play Store apps is allowed or disallowed
- The device's Android ID, IMSI, IMEI, phone number, and IP addresses
- Names of all apps installed on the device
- Contact list entries
- Call logs
- Names of all files stored in external storage

When HTTPS is used, the app deliberately disables checking of the server's certificate and hostname, thus enabling an attacker to easily perform a man-in-the-middle attack to intercept or manipulate communication. The app also sends some data to the same server using HTTP. HTTP provides no cryptographic protection over the network, so interception or manipulation of communication is even simpler.

Various gathered data is written to internal storage with deliberately insecure file permissions (deliberately set to world readable and writable) or written to external storage (where it can be read or written by other apps through USB debugging, or potentially through physical access to the device SD card if applicable).

The app contains a broadcast receiver called SMSReceiver used to gather SMS messages received by the device. The Android OS broadcasts received SMS messages to all broadcast receivers with an intent-filter for "android.provider.Telephony.SMS_RECEIVED." To test the ability of app vetting tools to detect this commonly found vulnerability, SMSReceiver deliberately fails to verify the permission of the sender and fails to check that the received intent's action string actually matches "android.provider.Telephony.SMS_RECEIVED." Therefore, a malicious app could inject fake SMS messages into UploadDataApp that were not actually received by the device.

Starting in Android 6.0, checking the intent's action string is sufficient, as "android.provider.Telephony.SMS_RECEIVED" was added to the list of protected broadcast action strings that can only be sent by the Android OS, not by third-party apps. Prior to Android 6.0, the app must ensure that the sender holds "android.permission.BROADCAST_SMS." Because apps are generally designed to run on a diverse array of Android versions, as a best practice any broadcast receiver for the SMS_RECEIVED action string should ensure the sender holds the BROADCAST_SMS permission.

The app contains a broadcast receiver called BootReceiver containing an intent-filter for the "android.intent.action.BOOT_COMPLETED" action string. A broadcast intent containing this action string is sent at device startup time. In order to test the ability of app vetting tools to detect a commonly found vulnerability, BootReceiver deliberately fails to check that the received intent's action string actually matches "android.intent.action.BOOT_COMPLETED." Therefore, a malicious app could inject an intent and trigger the BootReceiver service's functionality.

The app additionally deliberately exports services (SendIntentService, LocationService, and RecordIntentService) that are meant for internal use only to test the ability of app vetting tools to detect this issue. Exporting these services introduces a security vulnerability by enabling other apps resident on the device to invoke the services directly.

The app embeds in its code a default username and password used for HTTP Basic Authentication to a remote server. The app also writes the username and password value in the clear to a file in the app's internal data directory and to a file in the device's external storage directory (/sdcard). Storing cleartext passwords on the device, even in the app's internal data directory, is generally considered poor security practice.

The app embeds in its code a default AES [Advanced Encryption Standard] key used to encrypt gathered data (the data is stored locally and transmitted to the remote server both in unencrypted and encrypted form). The app does not follow cryptographic best practices for AES-CBC [Cipher Block Chaining] encryption: it uses a static initialization vector (also embedded in the code) instead of a randomly generated initialization vector, and the ciphertext is not authenticated (no Message Authentication Code (MAC) operation is applied to it).

The app demonstrates use of an unsupported platform API by using reflection to invoke the internal method com.android.internal.telephony.GsmAlphabet.stringToGsm7BitPacked.

The app's behavior is triggered automatically at app start time. For future work, as a more sophisticated test of the ability of vetting solutions to analyze app behavior, the app could be modified to delay its behavior for a set period of time or until triggered by a specific user interaction.

4.1.2 Custom-class-loader

This app would allow a malicious app to potentially bypass app vetting by downloading and executing new code after installation time (3J / NIAP FPT_AEX_EXT.1.4 and 4D / NIAP FPT_TUD_EXT.1.4). Because the new code is not included in the distributed application package, it will not be found through static analysis, but could potentially be found through dynamic analysis. An analysis solution might not identify the specifics of the malicious behavior in this case, since an adversary could dynamically adapt and target the downloaded code, causing different payloads to be delivered to different endpoints. An analysis solution should at least be able to detect that the app executes dynamic code downloaded after installation time and report the potential for abuse.

At app start time, this app connects to a remote server, downloads code, and then dynamically executes the downloaded code, both Dalvik (e.g., compiled Java) code and native code. The default downloaded code included in the public-released version of this app is benign. However, the MITRE team also prepared an alternate version of the downloaded code that includes much of the behavior described above in the UploadDataApp in order to perform a more advanced test of the ability of vetting solutions to analyze malicious app behavior. As described in section 4.1, UploadDataApp's malicious behavior is embedded in the compiled Android app package (APK) itself, where it can be detected through static analysis. With custom-class-loader, the malicious behavior can be downloaded from a remote server at runtime and not actually be in the app package itself, making it far more difficult for an analysis solution to detect the actual malicious behavior.

The app can be configured to download the code over either HTTP or HTTPS. When HTTPS is used, checking of the server's certificate and hostname is disabled by default (3D / NIAP FIA_X509_EXT.1.1, FCS_TLSC_EXT.1.2, FCS_TLSC_EXT.1.3 and 4C / NIAP FDC_DEC_EXT.1.4). The downloaded code is stored insecurely with world readable and writable file permissions by default, enabling malicious apps to overwrite the code (3C / NIAP

FDP_DAR_EXT.1.1) and also allowing the code to be overwritten via other vectors such as the Android Debug Bridge.

The app also includes native code in a library (liblocal_jni.so) bundled in the APK. The native code maps memory at an explicit address (3F / NIAP FPT_AEX_EXT.1.1) with both write and execute permissions (3G / NIAP FPT_AEX_EXT.1.2) to demonstrate violation of those two NIAP App PP requirements. The native code is compiled with stack protections deliberately disabled using the –fno-stack-protector compiler flag (3K / NIAP FPT_AEX_EXT.1.5).

4.1.3 Device Administrator Sample App

This sample app, extracted from the ApiDemos sample code in the Android Software Development Kit (SDK), demonstrates the use of Android's device administrator APIs (4H / no directly relevant NIAP requirement). Upon obtaining device administrator access (which requires user approval), apps can perform sensitive operations such as setting device security policies or wiping all apps and data stored on the device. Malicious Android apps have abused device administrator access in the past; for example, Lookout describes a ransomware family of apps called ScarePakage[8] and as previously described in section 3, Palo Alto Networks identified a family of Android malicious apps called Xbot.

4.1.4 Android Vulnerability Test Suite

This open source app, developed by NowSecure, assesses an Android device's susceptibility to various well-known public vulnerabilities. Although it technically does not perform malicious operations, it performs various system calls or other suspicious-looking operations based on public exploits (4G / NIAP AVA_VAN.1.2E). Examples of vulnerabilities that Android VTS checks for include:

- CVE-2011 1149 / PSNeuter / Ashmem Exploit
- CVE-2013-6282 / put/get_user
- CVE-2014-3153 / Futex bug / Towelroot
- Jar Bug 13678484 / Android FakeID
- Samsung WifiCredService remote code execution
- Stagefright bugs
- StumpRoot
- X.509 Serialization bug
- ZipBug 8219321 / Masterkey

4.1.5 Subterfuge App

The Subterfuge app is a commercially available game. Version 493 of the app contains the OpenSSL version number 1.0.1o which is susceptible to CVE-2015-1793. It is used to see if the app vetting solutions are capable of identifying third party library use and if so, whether or not

[8] https://blog.lookout.com/blog/2014/07/16/scarepakage/ (Accessed 20 January 2016.)

they list known vulnerabilities. This app was downloaded from the Play Store in binary format without source code available.

4.2 iOS App

4.2.1 Acme Airlines

This app demonstrates a multitude of vulnerable or potentially malicious activities that trace back to the evaluation criteria in section 3 (aligned with the NIAP App PP but including additional criteria not found in the PP). The app is meant to look like an electronic flight bag. This app demonstrates:

- Access to device hardware resources (4A / NIAP FDP_DEC_EXT.1.1) and to sensitive information repositories on the device (4B / NIAP FDP_DEC_EXT.1.2).
- Insecure writing of sensitive application data to device storage (3C / NIAP FDP_DAR_EXT.1.1 and FMT_CFG_EXT.1.2), and insecure network communication (3D / NIAP FDP_DIT_EXT.1.1, FIA_X509_EXT.1.1, FCS_TLSC_EXT.1.2, FCS_TLSC_EXT.1.3, 4C / FDP_NET_EXT.1.1).
- Inclusion of default credentials (3E / NIAP FMT_CFG_EXT.1.1) and insecure storage of credentials (3B / NIAP FCS_STO_EXT.1.1).
- Failure to invoke an appropriate random number generator where needed (3A / NIAP FCS_RBG_EXT.1.1) and other inappropriate cryptographic practices (3M).
- Use of an unsupported platform API (4E / NIAP FPT_API_EXT.1.1).
- The ability for a malicious app to potentially bypass app vetting by downloading and executing new code after installation time (3J / NIAP FPT_AEX_EXT.1.4 and 4D / NIAP FPT_TUD_EXT.1.4).

At app start time, the app demonstrates it has established access to device hardware resources by attempting to activate the device microphone for 5 seconds and sending the recorded audio to a remote server. It also uses iOS APIs to attempt to gather information from sensitive information repositories on the device and send the gathered information to a remote server via HTTPS (it can also be configured to use HTTP). This information includes the names of all apps installed on the device and contact list entries.

The app additionally attempts to gather the device's physical location (e.g., via GPS) once the user has clicked on a "Map" navigation menu and to send the location data to a remote server.

When HTTPS is used, the app deliberately disables checking of the server's certificate and hostname, thus enabling an attacker to easily perform a man-in-the-middle attack to intercept or manipulate communication. The app also sends some data to the same server using HTTP. HTTP provides no cryptographic protection over the network, so interception or manipulation of communication is even simpler. The app deliberately disables iOS App Transport Security so that it can disable HTTPS server certificate checking and so that it can use plaintext HTTP.

The app writes various gathered data to internal storage with deliberately insecure file permissions written using NSFileProtectionNone. The app also demonstrates Uniform Resource Identifier (URI) scheme hijacking by registering URI schemes used by the Google Chrome iOS app. This results in links from within Google apps to open in the AcmeAirlines app instead of

the Chrome app, which could be used by a malicious app to request credentials from a user as detailed in this article from FireEye.[9]

The app embeds in its code a default username and password used for HTTP Basic Authentication to the remote server. The app also writes the username and password value in the clear to a file in the app's internal data directory.

Like the UploadDataApp, this app similarly embeds in its code a default AES key used to encrypt gathered data which does not follow cryptographic best practices.

As a more sophisticated test of the ability of vetting solutions to analyze app behavior, the app has a time-bomb that is triggered to execute seven days after installation. The first time the user runs the app seven days after installation, it will gather calendar entries on the device and send them to the remote server.

This app also demonstrates the ability for a malicious app to potentially bypass Apple's app vetting via mobile code. The app utilizes a third party library called JSPatch to download a JavaScript "hot patch" file. The JavaScript is then converted to Objective-C code and executed using "method swizzling". Method swizzling exploits the ability to change a method's execution at runtime and can be used to modify an app's behavior dynamically. The app downloads a JavaScript hot patch file that injects use of the *allApplications* selector from the LSApplicationWorkspace class, a private API call, to obtain a list of apps installed on the device[10]. This demonstrates how a bad actor could use hot patching to introduce functionality that the App Store may not catch[11].

The iOS app does not address the following evaluation criteria items. For the reasons described in Appendix A.2, MITRE recommends that the iOS tests for these requirements be removed from the NIAP PP.

- 3F / NIAP FPT_AEX_EXT.1.1 – Assess ability to identify mapping of memory at explicit locations
- 3G / NIAP FPT_AEX_EXT.1.2 – Assess ability to identify mapping of memory as both writable and executable.
- 3K / NIAP FPT_AEX_EXT.1.5 – Assess ability to identify code compiled without stack-based buffer overflow protection enabled.

[9] https://www.fireeye.com/blog/threat-research/2015/02/ios_masque_attackre.html
[10] http://www.andreas-kurtz.de/2014/09/malicious-apps-ios8.html
[11] https://www.fireeye.com/blog/threat-research/2016/01/hot_or_not_the_bene.html

5 Tool Evaluation

MITRE conducted a market analysis of mobile app vetting tools and found 30 such commercial or open source offerings. The enclosed *App Vetting Tools Market Analysis* spreadsheet contains the findings of the market analysis. Given the limited resources of time and funding, MITRE evaluated only those tools most likely to satisfy the largest number of identified criteria. As such, the evaluation gave preference to tools that claimed conformance with the NIAP PP; the team also looked to Gartner reports[12][13] to select top performers.

5.1 Vetting Tools Examined

The tools evaluated were Android Lint (Included in Android Studio and Android SDK) and 8 other commercial products that have been anonymized due to non-disclosure agreements between MITRE and their respective companies. The products are identified by Product 1 – 8. Appendix C contains more information about Android Lint.

5.2 Findings

Overall, MITRE found that the best solutions generally employed a combination of static and dynamic analysis techniques. Static analysis techniques provide insights into the properties of the app and can detect many vulnerabilities, while dynamic analysis techniques reveal app behaviors that only occur at runtime.

Solution capabilities vary widely. Some solutions focus on traditional Java or other language coding issues and have limited coverage of weaknesses specific to high-risk Android and iOS mobile apps. Other solutions have a strong focus on specific weaknesses that commonly occur in Android and iOS apps. Therefore, enterprises should carefully assess capabilities and choose the best solution or solutions to meet their particular needs.

Some solutions can identify both vulnerabilities and potentially malicious or privacy-violating behaviors. However, in the solutions examined MITRE found that it would be trivial for a malicious app to identify the presence of an analysis environment versus a typical mobile device being used by a real user, and to adapt its behavior accordingly to evade detection. Even the solutions that use real mobile devices rather than emulators contain obvious indicators of an analysis environment, such as the presence of the Xposed framework (Android) or the Cydia app store (iOS) in the list of installed apps. Identifying vulnerabilities and identifying potentially malicious behaviors represent different use cases, and robustly detecting malicious applications is very difficult.

[12] Gartner's Application Security Testing Magic Quadrant 2015
[13] Gartner's Critical Capabilities for Application Security Testing 2015 – Mobile App Testing

Table 3 shows a stoplight chart comparing the criteria satisfied, partially satisfied, and not satisfied by the Android tools.

Table 3. Android Test Results

Assessment Criteria	Android Lint	Product 1	Product 2	Product 3	Product 4	Product 5	Product 6	Product 7	Product 8
3A Static IV for Encryption									
3B Cleartext Password File Storage									
3C Insecure Internal File Storage									
Insecure External File Storage									
3D Report Network Destinations and Ports									
Sensitive Data Cleartext									
Certificate Checking & Hostname Verify									
3E Embedded Default Credentials									
3F Memory Mapping Explicit Locations									
3G Memory Mapping Write and Execute									
3H Latest OS Anti-exploitation									
3J Executable Code Storage									
3K Stack-based Buffer Overflow Protection									
3L Identify 3rd Party Libraries									
3M Other Crypto Issues									
3N Inter-app Communication Security									
Issues									
4A Device Resource Permissions									
4B Sensor Access									
Sensitive Information Acess									
4D Dynamic Code Execution									
4E Use of Private/Unsupported APIs									
4F Obfuscation Detection									
4G Identify Known Malicious Code									
4H Device Administrator Access									
5A Detect Analysis Environment									
5B Multi-tenant Concerns									
6A Output formats									
6B Provide Evidence of Findings									
6C Enterprise Integration capabilities									

Table 4 shows the comparable results for iOS.

Table 4. iOS Test Results

Assessment Criteria	Product 2	Product 3	Product 4	Product 5	Product 6	Product 7	Product 8
3A Static IV for Encryption							
3B Cleartext Password File Storage							
3C nsecure Internal File Storage							
3D Report Network Destinations and Ports							
Sensitive Data Cleartext							
Certificate Checking & Hostname Verify							
3E Embedded Default Credentials							
3H Latest OS Anti-exploitation							
3L Identify 3rd Party Libraries							
3M Other Crypto Issues							
3N / inter-app Communication Security							
4A Device Resource Permissions							
4B Sensor Access							
Sensitive Information Acess							
4D Dynamic Code Execution							
4E Use of Private/Unsupported APIs							
4F Obfuscation Detection							
5A Detect Analysis Environment							
5B Multi-tenant Concerns							
6A Output formats							
6B Provide Evidence of Findings							
6C Enterprise Integration capabilities							
7A Unsanitized Input							
7B Code Coverage							

During the analysis, MITRE found that tools rarely supported the following criteria described in Section 3:

- 3A / NIAP FCS_RBG_EXT.1.1 – supported by only one tool
- 3E / NIAP FMT_CFG_EXT.1.1 – supported by only two tools
- 3F / NIAP FPT_AEX_EXT.1.1 – supported by no tools
- 3G / NIAP FPT_AEX_EXT.1.2 – supported by no tools
- 3K / NIAP FPT_AEX_EXT.1.5 – supported only partially by one tool
- 3M / crypto issues – supported by no tools

- 4G / NIAP AVA_VAN.1.2E – supported only partially by one tool
- 4H / Android Device Admin access – supported by only one tool
- 5A / Detection of analysis environment – supported only partially by one tool on iOS only
- 7B / Code coverage (iOS) – supported by no tools

6 Future Work

For future work in identifying potentially malicious apps, MITRE would like to explore the capabilities of mobile app intelligence services that focus on examining the metadata associated with each app rather than (or in addition to) the app's code or behavior itself. These services generally have access to data from the mainstream app stores as well as from a large set of real-world mobile devices. This appears as #2 in the criteria described in section 3, but the MITRE team did not have time to thoroughly examine product capabilities. Based on MITRE's brief analysis of capabilities, the team believes these services could provide a strong indication of the likely risk associated with each app. Potential data items analyzed by these solutions could include:

- Does this particular app (e.g., with the same hash or developer certificate) exist in the mainstream app stores or not? If it does, how popular is it (how many downloads, and what are the overall review scores)?
- How many apps exist from the same developer in mainstream app stores? How popular are they? Have security issues been identified in other apps from the same developer?
- Do other apps exist (either in the mainstream app stores or on other mobile devices) with the same name as the app but signed by different developers?

MITRE has also outlined recommendations to NIAP detailing changes to the PP that streamline the process. These are listed in Appendix B. Enterprises may be able to utilize these recommendations to tailor their app vetting process, reducing unneeded manual inspection of apps.

Appendix A Android Platform Mitigations

The Android OS includes built-in mitigations that address parts of the vulnerable and malicious behavior described above. The subsections below describe some of the mitigations. MITRE could expand this section for future work, and recommends adding a section describing Apple iOS built-in mitigations, many of which are described in Apple's iOS Security Guide[14].

A.1 Access to Hardware Resources and Sensitive Information Repositories (NIAP App PP FDP_DEC_EXT.1.1 and FDP_DEC_EXT.1.2)

Android apps must request permission to access device hardware resources and sensitive information repositories. If the application does not have permission, the operating system does not allow access. These permission requests are placed in the application's AndroidManifest.xml file, which can be easily inspected by an app analysis solution. Analysts should ensure that permission requests are consistent with the purpose of the application.

On Android 6.0 (Marshmallow) and above, if the app targets API level 23 or higher (the targetSdkVersion field in the app's AndroidManifest.xml file), the app must additionally explicitly request each permission from the user at runtime the first time the permission is required. On Android 6.0 and above devices, regardless of the targeted API level, the device user can selectively disable individual app permissions at any time. As devices migrate to Android 6.0 and above and apps migrate to API level 23 and above, it will become more difficult for malicious applications to obtain improper permissions. More information about Android's runtime permission capabilities can be found at:

- Android Developers web site: Requesting Permissions at Runtime
  - https://developer.android.com/training/permissions/requesting.html
- What's new in Android security (M and N Version) Google I/O 2016 video
  - https://www.youtube.com/watch?v=XZzLjllizYs

Figure A-1 shows screenshot examples of Android's runtime permissions system. The screenshot on the left shows an app[15] requesting a permission from the device user at runtime. The screenshot on the right shows the device user's ability to selectively disable or enable individual app permissions through the device settings.

[14] https://www.apple.com/business/docs/iOS_Security_Guide.pdf
[15] Google's RuntimePermissions sample application, available at https://github.com/googlesamples/android-RuntimePermissions (retrieved 14 June 2016)

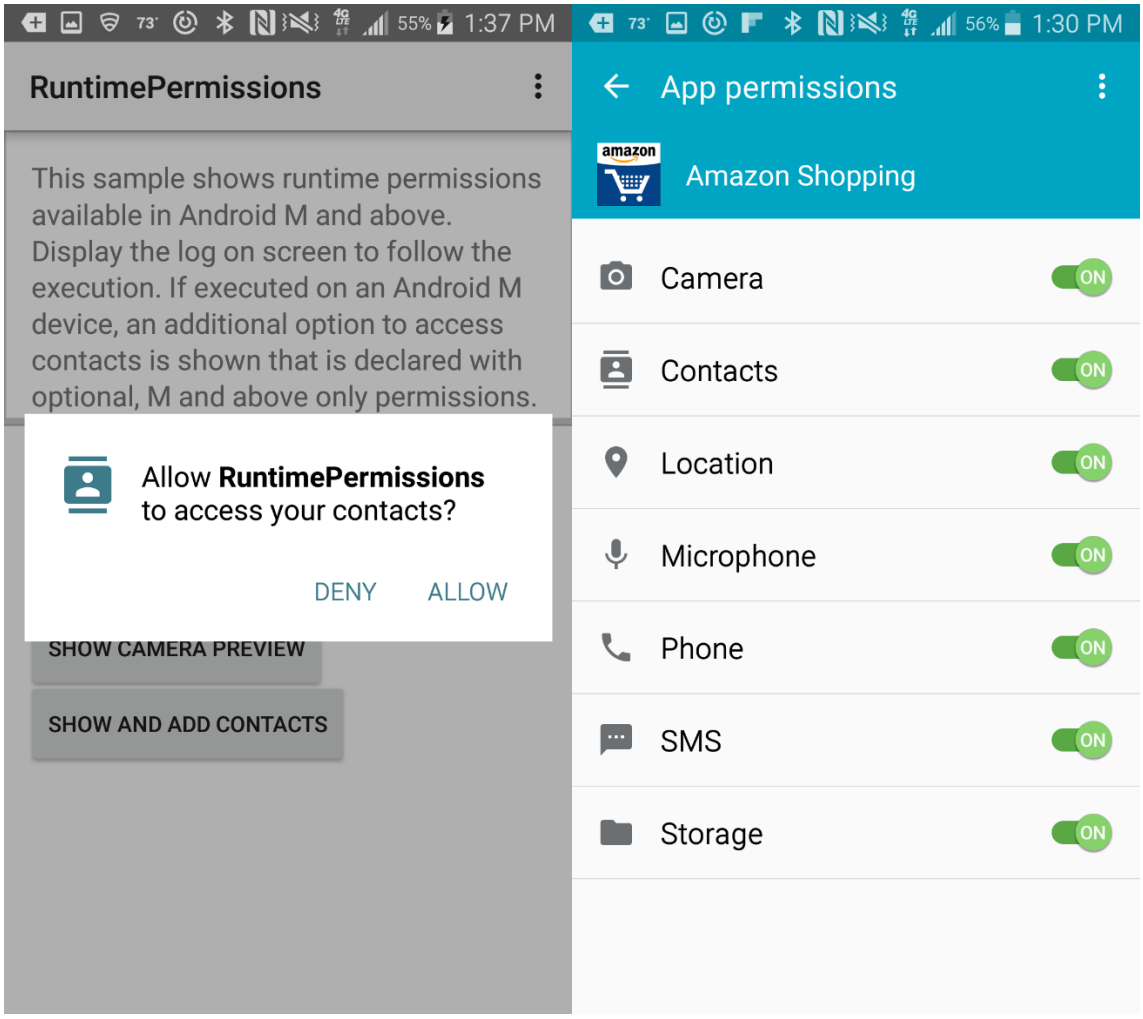

Figure A-1. Android's Runtime Permissions

When the Google Play Store is used to install applications on Android versions prior to 6.0, or for applications that do not target API level 23 or higher, it presents users at install-time with a list of requested permissions. The user must either accept all of the requested permissions or decline installation of the app. Figure A-2 provides an example of a permission request displayed by the Google Play Store at app installation time:

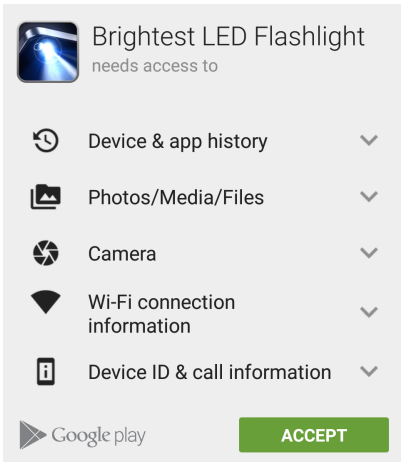

Figure A-2. Old Android Permissions

Samsung KNOX Workspace and Android for Work can provide additional mitigations against undesired access to sensitive information repositories. These apps can divide the device into separate personal and work-managed areas. They can give the user liberal control over the apps that can be installed in the personal area, while the enterprise can maintain full control over app installation in the work-managed area. Depending on configured policy, each area could, for example, be given its own contact list and calendar, with no ability to view the data in the other area. A malicious app running in the personal area of the device could not view the contact list and calendar in the work-managed area of the device.

A.2 Sensitive Application Data / File Permissions (FDP_DAR_EXT.1.1 and FMT_CFG_EXT.1.2)

Each Android app has its own internal storage directory that by default cannot be accessed by other apps. Android has deprecated the ability to set insecure (world readable or world writable) file permissions on files in internal storage since Android 4.2 (released in October 2012). Starting in Android N Developer Preview 2,[16] the ability to set insecure file permissions through Android SDK calls is prohibited for apps targeting Android N or above (an exception is thrown if such an attempt is made), and the app's internal storage directory is given UNIX file permission 700 by default, preventing other apps from accessing its contents regardless of the permissions of individual files. Vulnerable apps can override these secure behaviors, but deliberate effort is required.

As part of a previous research effort,[17] MITRE recommended to the Android Open Source Project that it enhance Android's SELinux policies to strictly prohibit apps from reading or writing other apps' internal data regardless of file permission. This recommendation has not been accepted, but the Android N Developer Preview 2 changes constitute an important step toward potentially enabling this stricter behavior in a future Android release. MITRE additionally strengthened the static analysis rules in the Android Software Development Kit (SDK)'s lint

[16] http://developer.android.com/preview/behavior-changes.html#permfilesys

[17] Additional details can be found at https://www.mitre.org/sites/default/files/publications/pr-16-0202-android-security-analysis-final-report.pdf

checker to help developers or security assessors detect inadvertent setting of insecure file permissions by apps.

Applications can still write to external storage, so care must be taken not to write sensitive data there.

Samsung KNOX Workspace and Android for Work can provide additional mitigations against insecure file storage. These apps can divide the device into separate personal and work-managed areas. They can give the user liberal control over the apps that can be installed in the personal area, while the enterprise can maintain full control over app installation in the work-managed area. Starting with Android 6.0 (and potentially earlier on Samsung KNOX devices), SELinux mandatory access controls prohibit apps running in one area from accessing data in the other area, regardless of file permissions. These controls apply to both internal and external storage, as the personal and work-managed areas are each given its own storage directory.

A.3 Network Communication (FDP_NET_EXT.1.1) / Protecting Data in Transit

Android's built-in TLS implementation defaults to secure practices; certificates are properly validated and hostnames are properly checked. However, app developers can inadvertently or deliberately override these practices and introduce vulnerabilities.

Android 6.0 introduced the "usesCleartextTraffic" flag in the AndroidManifest.xml file. It currently defaults to true. The app developer can explicitly set it to false if the app is not expected to use any cleartext network protocols; the OS then makes a best effort to only allow use of encrypted protocols such as TLS / HTTPS. However, as a best effort mechanism, this is not foolproof, and is only enforced if a TLS / HTTPS implementation that explicitly checks the flag is used (such as the Android OS built-in APIs). The Android N Developer Preview takes this capability further with the new Network Security Configuration feature[18] that enables apps to declare other data-in-transit security attributes; for example, to enable certificate pinning without the need for the app developer to write custom error-prone TLS certificate validation code. More details can be found in a post at the Android Developers Blog[19] and in the "What's new in Android security (M and N Version)" presentation at Google I/O 2016.[20]

[18] http://developer.android.com/preview/features/security-config.html
[19] https://android-developers.blogspot.com/2016/04/protecting-against-unintentional.html
[20] https://www.youtube.com/watch?v=XZzLjllizYs

Appendix B NIAP Protection Profile Recommendations

This appendix details areas where the NIAP Protection Profile for Application Software requirements could potentially be streamlined to better enable automation.

FPT_TUD_EXT.1.3: Assess ability to determine that all traces of the application have been deleted after uninstall.

MITRE recommends removing this test for iOS apps. On iOS, this test may be impractical to perform, as it is not possible to search the entire filesystem without a jailbroken device. Additionally, the test appears unnecessary on iOS, as the operating system should automatically remove the app's data when the app is uninstalled.

MITRE recommends skipping this test for any Android app that does not hold the WRITE_EXTERNAL_STORAGE permission. On Android, when an app is uninstalled, the operating system automatically deletes the app's internal storage directory as well as the app's directory on external storage, as described at http://developer.android.com/guide/topics/data/data-storage.html under the "Saving files that are app-private" section. The only other location to which apps are allowed to write is the general external storage (the /sdcard directory or its equivalent), and apps are only allowed to write there if they hold the WRITE_EXTERNAL_STORAGE permission.

Proposed Android text:

For Android, if the application's AndroidManifest.xml file does not contain a <uses-permission> or <uses-permission-sdk-23> tag containing android:name="android.permission.WRITE_EXTERNAL_STORAGE", then it is not necessary to perform any tests for this requirement. If the AndroidManifest.xml does contain such a tag, then the filesystem search can be limited to the device's external storage directories.

FPT_AEX_EXT.1.1: Assess ability to identify mapping of memory at explicit locations

MITRE recommends removing this test from the NIAP App PP's iOS Assurance Activity, as this vulnerability is unlikely to occur in an exploitable manner in iOS applications due to the protections against placing executable code in heap memory, as described in the next item.

FPT_AEX_EXT.1.2: Assess ability to identify mapping of memory as both writable and executable.

MITRE recommends removing this test from the NIAP App PP's iOS Assurance Activity, as this vulnerability is unlikely to occur in iOS applications due to protections in the iOS security architecture. As described on page 20 of the May 2016 version of the iOS Security Guide,[21] "[m]emory pages marked as both writable and executable can only be used by apps under tightly controlled restrictions: The kernel checks for the presence of the Apple-only dynamic code-signing entitlement. Even then, only a single mmap call can be made to request an executable and writable page, which is given a randomized address. Safari uses this functionality for its JavaScript JIT compiler."

FPT_AEX_EXT.1.5: Assess ability to identify code compiled without stack-based buffer overflow protection enabled.

[21] https://www.apple.com/business/docs/iOS_Security_Guide.pdf (retrieved June 13, 2016)

MITRE recommends removing this test from the NIAP App PP's iOS Assurance Activity. Xcode, the development environment used by iOS app developers, compiles code with stack protection enabled by default. In MITRE's experimentations, the team could not find a way to disable stack protection in Xcode.

Appendix C Tool Assessments

C.1 Android Lint

Android Lint is a free, open source tool primarily written by Google, and integrated into the SDK and Android Studio IDE. Given this integration into these environments commonly used by Android app developers, Android Lint enables app developers to easily find and correct issues early in the app development lifecycle.

Android Lint performs static analysis of apps (primarily Java source, Java bytecode, the AndroidManifest.xml, and app resource XML files) in search of both security-related and non-security-related issues. MITRE contributed several new security checks to the tool as part of a Fiscal Year 2015 research effort, described further in a separate publicly released technical report.[22]

Android Lint satisfied eight of the criteria in section 3 fully and one partially.

Figure C-1 shows Android Lint's integration into Android Studio. In the screenshot, Android Lint identified an issue in the app's TrustManager class that makes the app's network communication susceptible to attack. The screenshot does not show the full reported details.

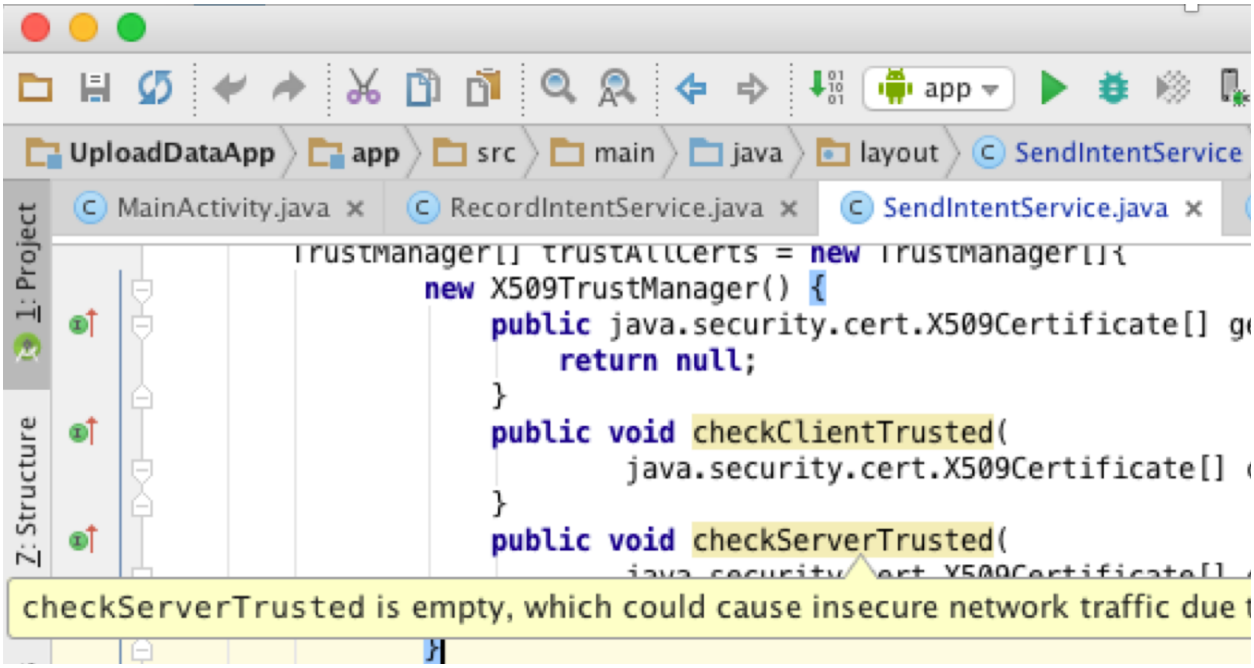

Figure C-1. Android Lint Screenshot

[22] https://www.mitre.org/sites/default/files/publications/pr-16-0202-android-security-analysis-final-report.pdf

Appendix D Acronyms

API	Application Programming Interface
APK	Android app package
CVE	Common Vulnerabilities and Exposures
DHS	Department of Homeland Security
DISA	Defense Information Systems Agency
EMM	Enterprise Mobile Management
FEDRAMP	Federal Risk Authorization Management Program
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
MDM	Mobile Device Management
MIT	Massachusetts Institute of Technology
NIAP	National Information Assurance Partnership
PP	Protection Profile
SDK	Software Development Kit
SMS	Short Message Service
TLS	Transport Layer Security
UI	User Interface
URI	Uniform Resource Identifier