NIST Internal Report
NIST IR 8462

Static Analysis Tool Exposition (SATE) VI: Mobile Track Report

Michael Ogata

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8462

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Internal Report
NIST IR 8462

Static Analysis Tool Exposition (SATE) VI: Mobile Track Report

Michael Ogata
Applied Cybersecurity Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8462

March 2023

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

NIST IR 8462
March 2023

Publication History
Approved by the NIST Editorial Review Board on 2023-01-04

How to Cite this NIST Technical Series Publication
Ogata M (2023) Static Analysis Tool Exposition (SATE) VI: Mobile Track Report. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Internal Report (IR) NIST IR 8462.
https://doi.org/10.6028/NIST.IR.8462

NIST Author ORCID iDs
Michael Ogata: 0000-0002-8457-2430

Contact Information
michael.ogata@nist.gov

Abstract

Mobile applications are pervasive in the public and private sectors. Enterprises in these sectors should evaluate the mobile applications used within their infrastructures for vulnerabilities to minimize potential risk. The SATE VI Mobile track seeks to improve the tools and services used in these evaluations by extending the Static Analysis Tool Exposition (SATE) to include mobile application tool evaluations. This document describes NIST's first attempt to carry out that goal and the results that stemmed from the first Mobile SATE track.

Keywords

cybersecurity, software quality, mobile applications, apps, vetting.

Table of Contents

1. Introduction 1
1.1. Background 1
1.2. Project Goals 1
1.3. Document Structure 1
2. Methodology 2
2.1. Mobile Security Application Platform (MSAP) Orchestration Platform: 3
2.2. Exercise Structure 3
2.2.1. Vendor Analysis Phase 4
2.2.2. Meta-Analysis Phase 4
2.3. Vulnerability Sources 5
2.3.1. NIAP Requirements for Vetting Mobile Apps from the Protection Profile for Application Software 5
2.3.1.1. Cryptographic Support (FCS) 6
2.3.1.2. User Data Protection (FDP) 7
2.3.1.3. Identification and Identification (FIA) 7
2.3.1.4. Security Management (FMT) 7
2.3.1.5. Protection of the TSF (FPT) 7
2.3.1.6. Trusted Paths/Channels (FTP) 8
2.3.2. Common Weakness Enumeration 8
3. Test Cases 9
3.1. Test Case 1 – VLC Multimedia Player 11
3.2. Test Case 2 – Forced Path Test 11
3.3. Test Case 3 – Native API Test 12
3.4. Test Case 4 – Reflection Test 12
3.5. Test Case 5 – Code Execution Demo App 13
3.6. Test Case 6 – Upload Data App 14
3.7. Test Case 7 – Device Admin Sample 17
4. Meta-Analysis and Results 17
4.1. Meta-Analysis by Test Case 18
4.1.1. Test Case 1 18
4.1.2. Test Case 2 19
4.1.3. Test Case 3 20
4.1.4. Test Case 4 21
4.1.5. Test Case 5 22
4.1.6. Test Case 6 23

4.1.7. Test Case 7 24
4.2. Results 25
4.2.1. Round Trip Time Per Test Case 25
4.2.2. Overall Identification Performance 27
5. Conclusions 28
5.1. Overall Performance 28
5.2. Lessons Learned 28
5.2.1. Formalized Report Submission Format 28
5.2.2. Mixing Vulnerability Sources 29
5.2.3. Expanding Test Case Resource Pool 29
References 30
Appendix A. List of Acronyms 31
Appendix B. Glossary 32

List of Tables

Table 1. Result Set Alias and #Tools per Result Set 3
Table 2. Meta-Analysis Methods 5
Table 3. Master Test Key 10
Table 4. Test Case 1 NIAP Vulnerabilities 11
Table 5. Test Case 1 CWE Vulnerabilities 11
Table 6. Test Case 2 NIAP Vulnerabilities 11
Table 7. Test Case 2 CWE Vulnerabilities 12
Table 8. Test Case 3 NIAP Vulnerabilities 12
Table 9. Test Case 3 CWE Vulnerabilities 12
Table 10. Test Case 4 NIAP Vulnerabilities 13
Table 11. Test Case 4 CWE Vulnerabilities 13
Table 12. Test Case 5 NIAP Vulnerabilities 14
Table 13. Test Case 5 CWE Vulnerabilities 14
Table 14. Test Case 6 NIAP Vulnerabilities 16
Table 15. Test Case 6 CWE Vulnerabilities 17
Table 16. Test Case 7 NIAP Vulnerabilities 17
Table 17. Test Case 7 CWE Vulnerabilities 17
Table 18. Test Case 1 Positive Tool Identifications 18
Table 19. Test Case 2 Positive Tool Identifications 19
Table 20. Test Case 3 Positive Tool Identifications 20
Table 21. Test Case 4 Positive Tool Identifications 21
Table 22. Test Case 5 Positive Tool Identifications 22
Table 23. Test Case 6 Positive Tool Identifications 23
Table 24. Test Case 7 Positive Tool Identifications 25
Table 25. Average Positive Identification by Test Type 28

List of Figures

Fig. 1. SATE Phases 4
Fig. 2. CWE Hierarchy 8
Fig. 3. Vulnerability Identification Test Case 1 18
Fig. 4. Vulnerability Identification Test Case 2 19
Fig. 5. Vulnerability Identification Test Case 5 20
Fig. 6. Vulnerability Identification Test Case 4 21
Fig. 7. Vulnerability Identification Test Case 5 22
Fig. 8. Vulnerability Identification Test Case 6 23
Fig. 9. Vulnerability Identification Test Case 7 24
Fig. 10. Round Trip Time 25
Fig. 11. Analysis Time by Test Case 26
Fig. 12. Misses and Successes Grouped by Vulnerability Name 27

Acknowledgments

The author of this document thanks the following collaborators:

- The Department of Homeland Security Science and Technology Directorate
- The MITRE Corporation
- Apcerto

1. Introduction

1.1. Background

Mobile apps are ubiquitous and have inexorably changed the way the federal government does business. Both federal organizations and their industry partners recommend mobile application vetting as a crucial component of an organization's cybersecurity stance to ensure mobile apps are free from software vulnerabilities [1].

There are many commercial solutions that seek to provide application security analysis as a service or a tool; however, tools vary greatly in their capabilities and domain knowledge. The Software Assurance Metrics and Tool Evaluation (SAMATE) project is a NIST project based in the Information Technology Laboratory [2]. The project's primary goal is to aide in the improvement of the state of the art in software assurance testing tools. To facilitate this, SAMATE team hosts the Static Analysis Tool Exposition (SATE) [3]. The core of the SATE activity is to invite static analysis tool vendors to participate in analyzing software with known vulnerabilities. The end goal of the activity is to measure the accuracy of those tools in successfully identifying the vulnerabilities. The SATE is currently on its 6th iteration.

1.2. Project Goals

In previous expositions, the SATE focused only on the performance of static analysis tools as they pertained to desktop and server software. This year's exposition seeks to expand the purview of the activity to include tools that evaluate mobile applications by hosting a Mobile Track targeting the Android mobile app ecosystem. To remain as open ended as possible, the mobile track expands scope of permitted software analysis tools by not restricting eligible tools to just those that carry out static analysis. This allowed for tools that implemented dynamic analysis techniques to participate. Furthermore, no restriction was made on the level of automation required in the app analysis, as various solutions implement varying levels of human interaction. For the purposes of the exercise, we treated the analysis methods implemented by the participants as a black box. The goal of the SATE mobile track, however, remains unchanged; to measure the performance of tools and services that assess mobile applications for security vulnerabilities in order to improve users' knowledge about tool capability in general and to help tool makers identify areas for improvement.

1.3. Document Structure

The remainder of this document is structured as follows:

- **Section 2** - Describes how the mobile track was structured and executed
- **Section 3** - Describes each of the test cases used as the basis for the mobile track
- **Section 4** - Summarizes the results of each test case
- **Section 5** - Discusses lessons learned and recommendations for the next iteration of the mobile track

2. Methodology

The goal of the SATE VI Mobile Track is to evaluate the effectiveness of tools to identify vulnerabilities in mobile apps. To accomplish this, we designed an activity to determine how many known vulnerabilities a tool could successfully identify from a series of mobile apps that intentionally included known vulnerabilities. More specifically:

- We selected a corpus of mobile apps that had known vulnerabilities. Collectively, the set of mobile apps is referred to as the test corpus from here on.
- Participants ran their tools against this corpus.
- The SATE team evaluated the results and presented them in this report.

Mobile app analysis tools vary in their capabilities and strategies [4]. A key distinguishing factor for how an analysis tool functions concerns what exactly the tool is actually examining. Some focus on just an application binary. That is, they interrogate the files that will end up resident on the mobile device itself. Others examine the source code of an application. Each of these strategies has its own merits and weaknesses, a deep dive into which is out of scope for this document. However, NIST SP 800-163 [1] goes into a detailed description into the variety of strategies employed by mobile app analysis tools.

To remain as solution-agnostic as possible, mobile apps were distributed to participants as units of both application binaries and source code. Together, these units are collectively referred to as a test case. Each test case was designed to express a known set of vulnerabilities. Section 2.2 goes into depth describing how we sourced vulnerabilities for inclusion in the activity. Section 3 details which vulnerabilities were included in each test case.

In the same way there is variability in the subject of analysis for mobile app analysis tools, there is also variability in the strategies employed by tools in how they conduct their analysis. Some solutions are completely autonomous, while others are human driven. Some tools can change the depth, time, and resources requirements for their evaluations depending on the parameters established by their customers.

The goal of the SATE mobile track is to measure the effectiveness of participant offerings. Each vendor was allowed to include as many of their supported solutions and/or solution configurations as desired. This allowed for participants to maximize their representation based on the full range of their capabilities Each unique instance of a solution is referred to as an *analysis tool*, or simply *tool*.

To participate, each tool was required to submit a report containing the vulnerability analysis for each test case. Again, to be as solution agnostic as possible, the only constraints placed on report format were:

- Each report must be submitted digitally
- Each report must be readable by the SATE team

The union of each set of reports associated with a single participant is referred to as a *result set* (see **Fig. 1**).

Seven participants participated in the SATE mobile track exercise. To preserve their anonymity, their results sets were given alphabetic aliases. **Table 1** summarizes this information:

Table 1. Result Set Alias and #Tools per Result Set

Result Set Alias	# Tools
A	1
B	1
C	1
D	1
E	1
F	1
G	2

2.1. Mobile Security Application Platform (MSAP) Orchestration Platform:

To facilitate the distribution of the test cases and the collection of test case analysis, the SATE team employed the Mobile Security Application Platform (MSAP). The MSAP is a version of the NIST open-source tool: AppVet[1]. The AppVet tool acts as a central hub to monitor the status of mobile apps being tracked by the system. It can integrate with several existing mobile app security scanning tools and has a documented Application Programing Interface (API) for new tools to be incorporated. For the purposes of the SATE, the MSAP tool was used to measure how long it took to analyze a given app, specifically the duration between test case download, and analysis upload (see Section 4.2.1).

2.2. Exercise Structure

The SATE VI Mobile Track was divided into two phases: the *vendor analysis phase* and the *meta-analysis phase*. Figure 1 captures the overall workflow for the exercise, and each phase is described in the next two sub-sections.

[1] https://csrc.nist.gov/projects/appvet/

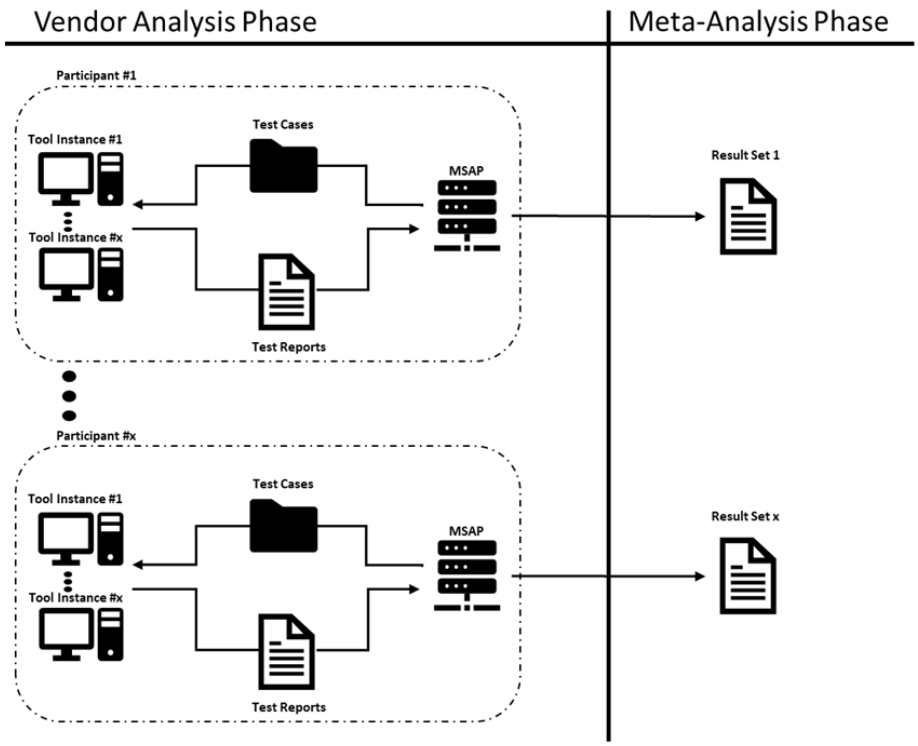

Fig. 1. SATE Phases

2.2.1. Vendor Analysis Phase

During the vendor analysis phase, each of the participating tools retrieved each of the test cases, analyzing them for vulnerabilities and submitting the results. Test case retrieval and result set submission was facilitated through the orchestration platform.

2.2.2. Meta-Analysis Phase

During the meta-analysis phase, the SATE team attempted to evaluate how many vulnerabilities were successfully identified in each result set. That is, we asked the following question. For a given test case, and a vulnerability known to be expressed in that test case, was a tool able to successfully indicate the vulnerability's existence. If the tool indicated existence, we defined it as a success.

Participating tools were required to submit an analysis report for each of the test cases. The minimal restrictions on the submission format lead to a lot of variability in these reports both in format (human vs machine readable) and in content (vocabulary, depth of detail, and supporting evidence). As such, the SATE Team used three strategies for identifying tool successes.

1. **Manual Evaluation** - A SATE team member read and extracted relevant statements from the result set, using a combination of keyword searching and context clue to establish success. Manual evaluation was utilized when the submitted reports were human readable.

2. **Automated Evaluation** - A keyword search was applied to a result set. Automated Evaluation was used to examine machine readable reports and to augment manual evaluation where it was deemed beneficial.
3. **Self-Evaluation** - After result set submission, vendors were provided with an answer key with the expected results. Vendors were allowed to self-attest to tool successes by mapping portions of their results to known vulnerabilities. This allowed vendors to make the case for their tool's performance and to account for any discrepancies/inconsistencies made by SATE team members during manual/automated evaluations. Note, not all vendors opted to participate in self-evaluation.

Table 2. Meta-Analysis Methods

Result Set Alias	Manual	Automated	Self-Evaluation
A	●		
B	●		●
C	●		
D		●	
E	●		
F	●	●	
G	●	●	●

To arrive at the final performance for each tool, a union was taken of each meta-analysis method available for that tool.

2.3. Vulnerability Sources

Measuring the effectiveness of identifying vulnerabilities is predicated on having a common vocabulary with which to describe them. NIST 800-163 Vetting the Security of Mobile Applications identifies multiple vocabularies for describing vulnerabilities [1]. The SATE team settled on two vocabularies for vulnerability identification:

- Requirements for Vetting Mobile Apps from the Protection Profile for Application Software [5]
- Common Weakness Enumeration (CWE) [6]

The remainder of this section provides a high-level description of each of these vocabularies.

2.3.1. NIAP Requirements for Vetting Mobile Apps from the Protection Profile for Application Software

The National Information Assurance Partnership (NIAP) is tasked with evaluating information technology products for conformance to the international Common Criteria [7]. To achieve this, the NIAP publishes Protection Profiles (PP) [8]. A PP defines a discrete list of attestations concerning the construction and behavior befitting an Information Technology (IT) production

that can be said to be compliant with the Common Criteria. IT products that are considered to be part of a National Security System are required by policy[2] to pass Common Criteria evaluation.

The NIAP defines a profile for general application software [9]. The PP describes not only how an application should be designed to be considered compliant but also the activities an evaluator can use to establish compliance.

Mobile apps often fall outside of the purview of what is part of a National Security System. Despite this, the NIAP recognizes the usefulness of evaluating mobile apps in terms the Application PP. To satisfy this need, the NIAP publishes a subset of the greater Application PP known as the Requirements for Vetting Mobile Apps from the Protection Profile for Application Software. For brevity, this document will refer to this subset as the NIAP Vetting Requirements.

The NIAP Vetting Requirements divide their security requirements into two sub-domains:

1. Security Functional Requirements – mobile app design, configuration, and behavior requirements
2. Security Assurance Requirements – describes methodology requirements placed on the actors evaluating the mobile app

For the SATE mobile track test cases, we utilized functional requirements. NIAP functional requirements are labeled using the following nomenclature:

$$\underbrace{FCS}_{Class}\_\underbrace{RGB\_EXT}_{Short\ Name}.\underbrace{1.1(1)}_{Variant}$$

The remainder of this subsection lists the functional requirements utilized in the SATE VI Mobile Track. For many of the functional requirements, compliance is determined situationally based on the behavior of the app. For example, FCS_RBG_EXT.1.1 describes how an app obtains a random number from a random number generator. If an app does not utilize random numbers, an evaluator can show compliance by establishing this. For more detailed descriptions of each of the requirements please see [5] and [8]. Sections 2.3.1.1 - 2.3.1.6 list each of the functional requirements included in the test cases.

2.3.1.1. Cryptographic Support (FCS)

- **FCS RBG EXT.1.1:** the application shall complete one of the following options for its cryptographic operations:
  - use no Deterministic Random Bit Generator (DRBG) functionality
  - invoke platform-provided DRBG functionality
  - implement DRBG functionality
- **FCS STO EXT.1.1:** When handling credentials, the application shall do one of the following:
  - not store any credentials
  - invoke the functionality provided by the platform to securely store credentials
  - implement functionality to securely store credentials to non-volatile memory
- **FCS TLSC EXT.1.1:** The application shall invoke platform-provided Transport Layer Security (TLS) 1.2 and implement TLS 1.2

[2] https://www.cnss.gov/CNSS/issuances/Policies.cfm

- **FCS TLSC EXT.1.2:** The application shall verify that the presented identifier matches the reference identifier according to RFC 6125 [10]
- **FCS TLSC EXT.1.3:** The application shall establish a trusted channel only if the peer certificate is valid

2.3.1.2. User Data Protection (FDP)

- **FDP DAR EXT.1.1:** When storing sensitive data, the application shall:
  - leverage platform-provided functionality to encrypt sensitive data
  - implement functionality to encrypt sensitive data
  - not store any sensitive data in non-volatile memory
- **FDP DEC EXT.1.1:** The application shall restrict its access to the following resources: network connectivity, camera, microphone, location services, near-field communication (NFC), Universal Serial Bus (USB), Bluetooth, etc. That is, the app does not access resources it does not explicitly require to function as intended.
- **FDP DEC EXT.1.2:** The application shall restrict its access to sensitive information repositories, address book, calendar, call lists, system logs, etc.
- **FDP NET EXT.1.1:** The application shall restrict network communication in the following ways:
  - permit no network communication
  - restrict user-initiated communication
  - restrict network communication the application can respond to

2.3.1.3. Identification and Identification (FIA)

- **FIA X509 EXT.1.1:** The application shall either invoke platform-provided functionality to handle X.509 certificates or, if not available, implement said functionality

2.3.1.4. Security Management (FMT)

- **FMT CFG EXT.1.1:** The application shall provide only enough functionality to set new credentials when configured with default credentials or no credentials
- **FMT CFG EXT.1.2:** The application shall be configured by default with file permissions that protect it and its data from unauthorized access

2.3.1.5. Protection of the TSF (FPT)

- **FPT AEX EXT.1.1**: The application shall not request to map memory at an explicit address except for explicitly defined purposes
- **FPT AEX EXT.1.2:** The application shall not allocate any memory region with both write and execute permissions unless the functionality is explicitly documented and accounted for
- **FPT AEX EXT.1.3:** The application shall be compatible with security features provided by the platform vendor

- **FPT AEX EXT.1.4:** The application shall not write user-modifiable files to directories that contain executable files unless explicitly directed by the user to do so
- **FPT AEX EXT.1.5:** The application shall be compiled with stack-based buffer overflow protection enabled. For applications using Java Native Interface (JNI), the evaluator shall ensure that the -fstack-protector-strong or fstackprotector-all flags are used. The -fstack-protector-all flag is preferred, but fstackprotector-strong is acceptable.
- **FPT API EXT.1.1:** The application shall use only documented platform API
- **FPT TUD EXT.1.4:** The application shall not download, modify, replace or update its own binary code

2.3.1.6. Trusted Paths/Channels (FTP)

- **FTP DIT EXT.1.1:** The application shall either not transmit any data, not transmit any sensitive data, or encrypt all transmitted sensitive data

2.3.2. Common Weakness Enumeration

The Common Weakness Enumeration is a list of recognized, real-world, software weakness. Maintained by the MITRE Corporation [6], CWE aims to provide a common language for describing software vulnerabilities. The CWE database groups CWEs into various layers, views, and subgroups. Furthermore, each CWE is assigned an abstraction level that describes how specific it is.

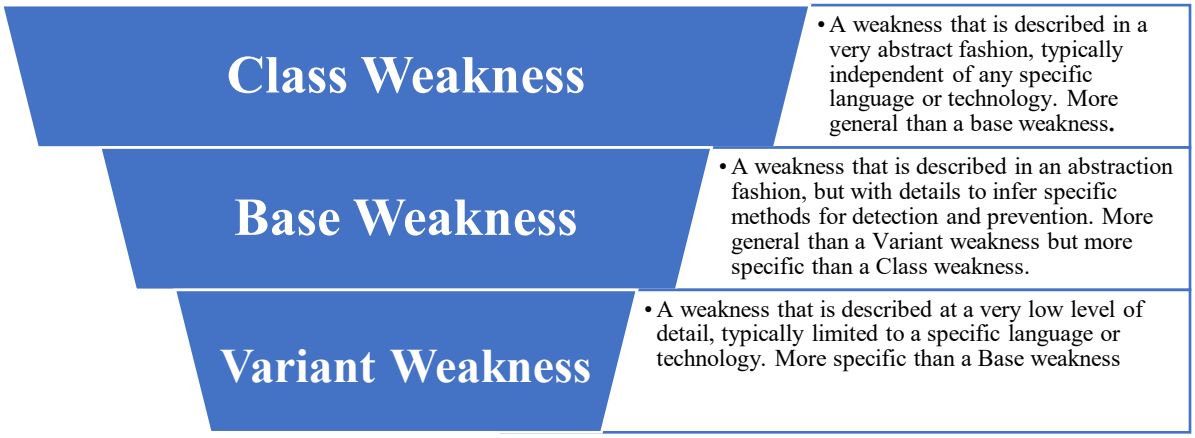

Fig. 2. CWE Hierarchy

The remainder of this section contains a list of the CWEs included in SATE VI Mobile track as well as high-level descriptions of each. Additionally, a Uniform Resource Locator (URL) leading to the detailed description maintained by MITRE is included along with each CWE.

- **CWE-20 (Class):** Improper Input Validation. The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program.
- **CWE-77 (Class):** Improper Neutralization of Special Elements used in a Command ('Command Injection').

- **CWE-111 (Base):** Direct Use of Unsafe JNI
- **CWE-117 (Base):** Improper Output Neutralization for Logs. The software does not neutralize or incorrectly neutralizes output that is written to logs.
- **CWE-284 (Class):** Improper Access Control. The software does not restrict or incorrectly restricts access to a resource from an unauthorized actor.
- **CWE-285 (Class):** Improper Authorization
- **CWE-287 (Class):** Improper Authentication. When an actor claims to have a given identity, the software does not prove or insufficiently proves that the claim is correct.
- **CWE-295 (Base):** Improper Certificate Validation
- **CWE-296 (Base):** Improper Following of a Certificate's Chain of Trust
- **CWE-312 (Variant):** Cleartext Storage of Sensitive Information
- **CWE-319 (Variant):** Cleartext Transmission of Sensitive Information
- **CWE-349 (Base):** Acceptance of Extraneous Untrusted Data with Trusted Data
- **CWE-395 (Base):** Use of NullPointerException Catch to Detect NULL Pointer
- **CWE-494 (Base):** Download of Code Without Integrity Check
- **CWE-502 (Variant):** Deserialization of Untrusted Data. The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.
- **CWE-532 (Variant):** Information Exposure Through Log Files
- **CWE-552 (Base):** Files or Directories Accessible to External Parties
- **CWE-572 (Variant):** Call to Thread run() instead of start(). The program calls a thread's run() method instead of calling start(), which causes the code to run in the thread of the caller instead of the callee
- **CWE-732 (Class):** Incorrect Permission Assignment for Critical Resource
- **CWE-922 (Class):** Insecure Storage of Sensitive Information
- **CWE-925 (Variant):** Improper Verification of Intent by Broadcast Receiver. The Android application uses a Broadcast Receiver that receives an Intent but does not properly verify that the Intent came from an authorized source. Another application can impersonate the operating system and cause the software to perform an unintended action.

3. Test Cases

We provided seven test cases for analysis in the SATE VI mobile track. Test Case 1 was developed by the SATE Team. Test Cases 2, 3, and 4 were donated to the project from the Department of Homeland Security. Test cases 5, 6, and 7 were sourced from a mobile app evaluation conducted by the MITRE corporation [11]. **Table 3**, below, contains the master key for each of the vulnerabilities contained in each of the test cases. For each test case (numbered columns), a vulnerability (named rows) appears if an “x” is noted in the corresponding table cell.

Table 3. Master Test Key

	Test Case #								Test Case #						
Vulnerability Name	1	2	3	4	5	6	7	**Vulnerability Name**	1	2	3	4	5	6	7
FCS_RBG_EXT.1.1						x		**CWE-20**			x			x	
FCS_STO_EXT.1.1						x		**CWE-77**	x						
FCS_TLSC_EXT.1.1						x		**CWE-111**					x		
FCS_TLSC_EXT.1.2					x	x		**CWE-117**	x						
FCS_TLSC_EXT.1.3	x				x	x		**CWE-284**			x	x		x	x
FDP_DAR_EXT.1.1					x	x		**CWE-285**					x	x	
FDP_DEC_EXT.1.1						x		**CWE-287**	x						
FDP_DEC_EXT.1.2			x			x		**CWE-295**		x		x		x	
FDP_NET_EXT.1.1				x		x		**CWE-296**		x			x	x	
FIA_X509_EXT.1.1	x				x	x		**CWE-312**		x				x	x
FMT_CFG_EXT.1.1						x		**CWE-319**				x		x	
FMT_CFG_EXT.1.2	x					x		**CWE-349**						x	
FPT_AEX_EXT.1.1					x			**CWE-395**	x						
FPT_AEX_EXT.1.2					x			**CWE-494**					x		
FPT_AEX_EXT.1.4					x			**CWE-502**	x						
FPT_AEX_EXT.1.5					x			**CWE-532**					x	x	
FPT_API_EXT.1.1						x		**CWE-552**						x	
FPT_TUD_EXT.1.4	x				x			**CWE-572**	x						
FTP_DIT_EXT.1.1						x		**CWE-732**		x			x		
								CWE-922						x	
								CWE-925	x						

The remainder of this section describes each of these test cases. Each subsection contains a short description of the test case's mobile app, a table containing which NIAP vulnerabilities are represented in the test case, and a table containing which CWE vulnerabilities are represented in the test case. The CWE table also lists the specific source file and line number(s) on which the associated weaknesses occur. Where available, hyperlinks are included that can be used to retrieve the mobile app's source code.

3.1. Test Case 1 – VLC Multimedia Player

Available from: https://www.nist.gov/document/satemobiletestcase20170001targz

VLC is an open-source media player developed by the VideoLAN Organization [12]. The SATE team used the VLC app as a target for manual vulnerability injection. That is to say, the team modified the source code to contain code flaws. VLC was chosen for the following reasons:

- It is a very popular application on the Google Play Store with more than 100 million installs [13].
- It has reasonably large code base (thousands of files)
- It represents reasonably complex behavior: network operations (streaming media), video/audio decoding file manipulation, etc.

Table 4. Test Case 1 NIAP Vulnerabilities

NIAP Vulnerabilities
FCS_TLSC_EXT.1.3
FIA_X509_EXT.1.1
FMT_CFG_EXT.1.2
FPT_TUD_EXT.1.4

Table 5. Test Case 1 CWE Vulnerabilities

CWE	SOURCE FILE PATH	LINE #
77	vlc-test-suite-test-suite/vlc-android/src/org/videolan/vlc/gui/video/benchmark/BenchActivity.java	393
117	vlc-test-suite-test-suite/vlc-android/src/org/videolan/vlc/gui/network/MRLPanelFragment.java	110
287	vlc-test-suite-test-suite/vlc-android/AndroidManifest.xml	53
395	vlc-test-suite-test-suite/vlc-android/src/org/videolan/vlc/gui/browser/BaseBrowserFragment.java	141
502	vlc-test-suite-test-suite/vlc-android/src/org/videolan/vlc/gui/video/VideoPlayerActivity.java	3296
572	vlc-test-suite-test-suite/vlc-android/src/org/videolan/vlc/gui/tv/browser/MediaSortedFragment.java	73, 83
925	vlc-test-suite-test-suite/vlc-android/src/org/videolan/vlc/BootupReceiver.java	41

3.2. Test Case 2 – Forced Path Test

Available from: https://www.nist.gov/document/satemobiletestcase20170002targz

This test attempts to stop analysis by tools that utilize forced path execution techniques. The application also implements a trivial textbox that must have the text 'yes' inserted in it, this detects automated tools that just run the application and blindly click all buttons to simulate user input.

Table 6. Test Case 2 NIAP Vulnerabilities

NIAP Vulnerabilities
None Identified

Table 7. Test Case 2 CWE Vulnerabilities

CWE	SOURCE FILEPATH	LINE #
295	SATE_Mobile_TestCase_2017_0002/source/app/src/main/java/com/ais/forcedpathtest/MainActivity.java	78, 96, 108
296	SATE_Mobile_TestCase_2017_0002/source/app/src/main/java/com/ais/forcedpathtest/MainActivity.java	97, 109
312	SATE_Mobile_TestCase_2017_0002/source/app/src/main/java/com/ais/forcedpathtest/MainActivity.java	80, 81, 82, 83, 85, 86
732	SATE_Mobile_TestCase_2017_0002/source/app/src/main/java/com/ais/forcedpathtest/MainActivity.java	40

3.3. Test Case 3 – Native API Test

Available from: https://www.nist.gov/document/satemobiletestcase20170003targz

This app performs Android API calls from native C++ code. As most android analysis tools don't have the functionality to analyze native code, this is a method for obscuring access to the certain API calls.

This application accesses the device's IMEI number by calling internal android APIs from the native C code. Due to the way permissions work, this still requires the *android.permission.READ_PHONE_STATE* permission. This permission, while deemed 'dangerous' by the android SDK, is not always flagged in analysis tools.

Table 8. Test Case 3 NIAP Vulnerabilities

NIAP Vulnerabilities
FDP_DEC_EXT.1.2

Table 9. Test Case 3 CWE Vulnerabilities

CWE	SOURCE FILEPATH	LINE #
20	SATE_Mobile_TestCase_2017_0003/source/app/src/main/java/com/ais/nativeapitest/MainActivity.java	17
284	SATE_Mobile_TestCase_2017_0003/source/app/src/main/java/com/ais/nativeapitest/MainActivity.java	31

3.4. Test Case 4 – Reflection Test

Available from: https://www.nist.gov/document/satemobiletestcase20170004targz

This test case demonstrates how to perform Android API calls by accessing the internal/private android API. These internal API calls are often not monitored. This application sends a test Short Messaging Service (SMS)/text message using the internal API. Due to how permissions work in the Android Operating System (OS), the *android.permission.SEND_SMS* is still required. While this permission is flagged by many analysis tools, this technique can be used to hide functionality in applications that already require this permission.

Table 10. Test Case 4 NIAP Vulnerabilities

NIAP Vulnerabilities
FDP_NET_EXT.1.1

Table 11. Test Case 4 CWE Vulnerabilities

CWE	SOURCE FILEPATH	LINE #
284	SATE_Mobile_TestCase_2017_0004/source/app/src/main/java/com/ais/reflectiontest/MainActivity.java	30
295	SATE_Mobile_TestCase_2017_0004/source/app/src/main/java/com/ais/reflectiontest/MainActivity.java	36,37,42,47,51,54,55,57,48,52,58
319	SATE_Mobile_TestCase_2017_0004/source/app/src/main/java/com/ais/reflectiontest/MainActivity.java	48,52,58

3.5. Test Case 5 – Code Execution Demo App[3]

Available from: https://github.com/mpeck12/custom-class-loader

This app would allow a malicious app to potentially bypass app vetting by downloading and executing new code after installation time (FPT_AEX_EXT.1.4 and 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. 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 Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPS). When HTTPS is used, checking of the server's certificate and hostname is disabled by default (FIA_X509_EXT.1.1, FCS_TLSC_EXT.1.2, FCS_TLSC_EXT.1.3 and 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 (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 Android Package Kit (APK). The native code maps memory at an explicit address (FPT_AEX_EXT.1.1) with both write and execute permissions (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 (NIAP FPT_AEX_EXT.1.5).

[3] The prose in this section was borrowed directly from Section 4.1.2 of Analyzing the Effectiveness of App Vetting Tools in the Enterprise [11]. It is included here as a convenience to the reader, with minor edits to remove superfluous cross references and background information.

Table 12. Test Case 5 NIAP Vulnerabilities

NIAP Vulnerabilities
FCS_TLSC_EXT.1.2
FCS_TLSC_EXT.1.3
FDP_DAR_EXT.1.1
FIA_X509_EXT.1.1
FPT_AEX_EXT.1.1
FPT_AEX_EXT.1.2
FPT_AEX_EXT.1.4
FPT_AEX_EXT.1.5
FPT_TUD_EXT.1.4

Table 13. Test Case 5 CWE Vulnerabilities

CWE	SOURCE FILEPATH	LINE #
111	custom-class-loader/app/src/main/java/com/example/dex/MainActivity.java	304
285	custom-class-loader/app/src/main/java/com/example/dex/MainActivity.java	177, 316, 387, 155
296	custom-class-loader/app/src/main/java/com/example/dex/MainActivity.java	225
494	custom-class-loader/app/src/main/java/com/example/dex/MainActivity.java	303, 311
532	custom-class-loader/app/src/main/java/com/example/dex/MainActivity.java	159
732	custom-class-loader/app/src/main/java/com/example/dex/MainActivity.java	368, 533, 297, 298, 302, 378, 536

3.6. Test Case 6 – Upload Data App[4]

Available from: https://github.com/mitre/uploaddataapp

This app demonstrates the following vulnerabilities:

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

[4] The prose in this section was borrowed directly from Section 4.1.2 of Analyzing the Effectiveness of App Vetting Tools in the Enterprise [11] . It is included here as a convenience to the reader, with minor edits to remove superfluous cross references and background information.

At app start time, the app demonstrates it has established access to device hardware resources by attempting to access both the device microphone and global positioning system (GPS) with the intent of transmitting information gathered from these to a remote server. The app also attempts to intercept Short Message Service (SMS) messages received by the device and send them to a remote server. Furthermore, when the app starts, it uses Android APIs to attempt to gather information from the device's sensitive information repositories and send the gathered information via HTTPS (it can also be configured to use HTTP) to a remote location. 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 Identifier (Android ID), international mobile subscriber identity (IMSI), International Mobile Equipment Identity (IMEI), phone number, and Internet Protocol (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 are 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 Secure Digital (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 Advanced Encryption Standard (AES) key used to encrypt gathered data (the data are 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.

Table 14. Test Case 6 NIAP Vulnerabilities

NIAP Vulnerabilities
FCS_RBG_EXT.1.1
FCS_STO_EXT.1.1
FCS_TLSC_EXT.1.1
FCS_TLSC_EXT.1.2
FCS_TLSC_EXT.1.3
FDP_DAR_EXT.1.1
FDP_DEC_EXT.1.1
FDP_DEC_EXT.1.2
FDP_NET_EXT.1.1
FIA_X509_EXT.1.1
FMT_CFG_EXT.1.1
FMT_CFG_EXT.1.2
FPT_API_EXT.1.1
FTP_DIT_EXT.1.1

Table 15. Test Case 6 CWE Vulnerabilities

CWE	SOURCE FILEPATH	LINE #
20	uploaddataapp/app/src/main/java/com/example/uploaddataapp/InjectSMSService.java	59, 133
284	uploaddataapp/app/src/main/java/com/example/uploaddataapp/RecordIntentService.java	51, 57, 66, 67, 73, 74, 75
284	uploaddataapp/app/src/main/java/com/example/uploaddataapp/LocationService.java	139, 140, 141, 142
285	uploaddataapp/app/src/main/java/com/example/uploaddataapp/RecordIntentService.java	63
295	uploaddataapp/app/src/main/java/com/example/uploaddataapp/InjectSMSService.java	83
296	uploaddataapp/app/src/main/java/com/example/uploaddataapp/InjectSMSService.java	100
312	uploaddataapp/app/src/main/java/com/example/uploaddataapp/LocationService.java	145
312	uploaddataapp/app/src/main/java/com/example/uploaddataapp/ListFiles.java	41, 47
319	uploaddataapp/app/src/main/java/com/example/uploaddataapp/RecordIntentService.java	80
319	uploaddataapp/app/src/main/java/com/example/uploaddataapp/LocationService.java	146, 159
349	uploaddataapp/app/src/main/java/com/example/uploaddataapp/InjectSMSService.java	115
532	uploaddataapp/app/src/main/java/com/example/uploaddataapp/RecordIntentService.java	58
552	uploaddataapp/app/src/main/java/com/example/uploaddataapp/ListFiles.java	28, 38, 46
922	uploaddataapp/app/src/main/java/com/example/uploaddataapp/RecordIntentService.java	71

3.7. Test Case 7 – Device Admin Sample[5]

Available from: https://github.com/mitre/device-admin-sample

This sample app, extracted from the ApiDemos sample code in the Android Software Development Kit (SDK), demonstrates the use of Android's device administrator.

Table 16. Test Case 7 NIAP Vulnerabilities

NIAP Vulnerabilities
None Identified

Table 17. Test Case 7 CWE Vulnerabilities

CWE	SOURCE FILEPATH	LINE #
284	device-admin-sample/app/src/main/java/com/example/deviceadminsample/DeviceAdminSample.java	118, 169, 170, 171, 172, 326, 385, 395, 397, 401, 405, 466, 467, 468, 469, 470, 471, 472, 473, 629, 870, 900, 1018, 1057, 1103
312	device-admin-sample/app/src/main/java/com/example/deviceadminsample/DeviceAdminSample.java	562, 956

4. Meta-Analysis and Results

This section contains the results to the SATE VI mobile track. Section 4.1 shows the results for each of the seven tests cases. Section 4.2 contains an analysis of the overall performance of tools in the exercises.

[5] The prose in this section was borrowed directly from Section 4.1.2 of Analyzing the Effectiveness of App Vetting Tools in the Enterprise [11]. It is included here as a convenience to the reader, with minor edits to remove superfluous cross references and background information.

4.1. Meta-Analysis by Test Case

The following subsections detail the findings for each of the included test cases. For each test case, we show how many tools were able to positively identify each of the test case's included vulnerabilities. As there were seven participating tools in the exercise, the maximum number of expected positive identifications for each vulnerability is also seven depicted in each sub-section's figure as a solid vertical line).

4.1.1. Test Case 1

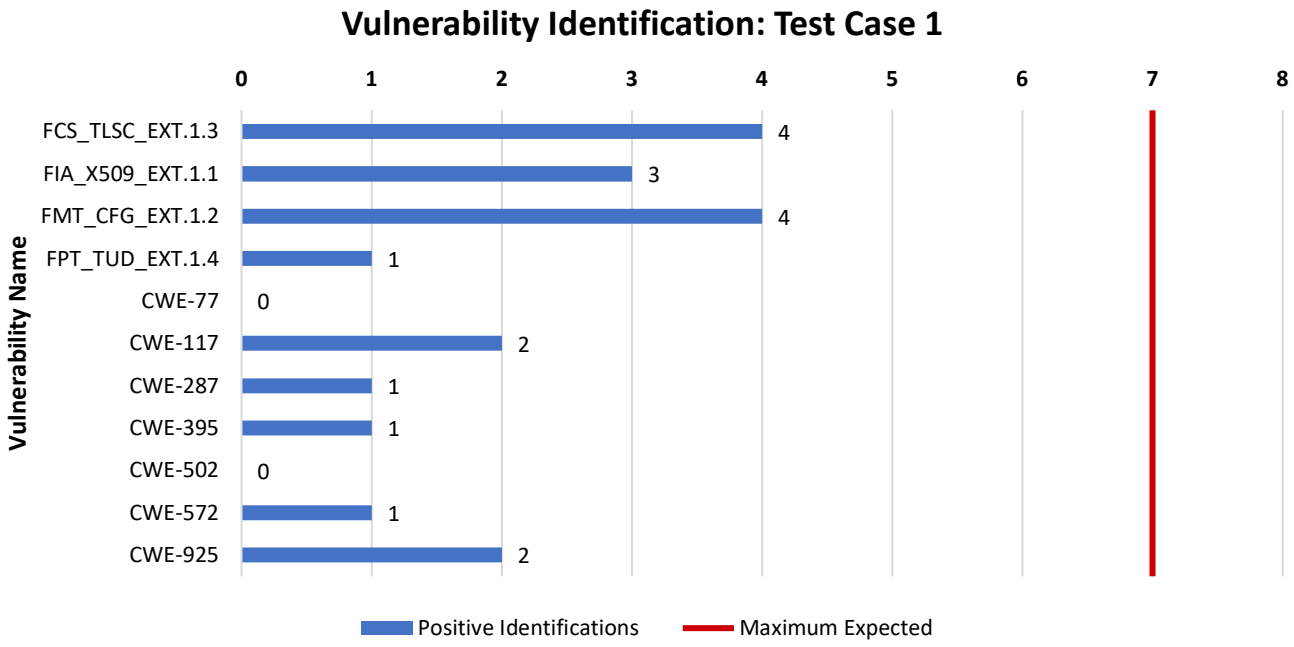

Fig. 3. Vulnerability Identification Test Case 1

Table 18. Test Case 1 Positive Tool Identifications

Vulnerability Name	# Positive Identifications
FCS_TLSC_EXT.1.3	4
FIA_X509_EXT.1.1	3
FMT_CFG_EXT.1.2	4
FPT_TUD_EXT.1.4	1
CWE-77	0
CWE-117	2
CWE-287	1
CWE-395	1
CWE-502	0
CWE-572	1
CWE-925	2

4.1.2. Test Case 2

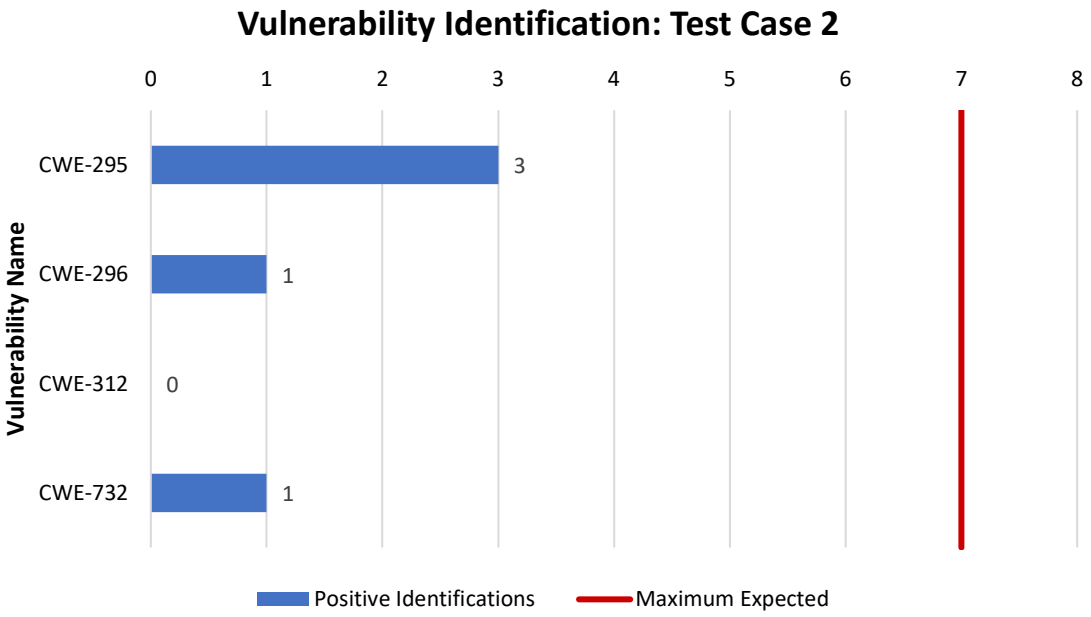

Fig. 4. Vulnerability Identification Test Case 2

Table 19. Test Case 2 Positive Tool Identifications

Vulnerability Name	# Positive Identifications
CWE-295	3
CWE-296	1
CWE-312	0
CWE-732	1

4.1.3. Test Case 3

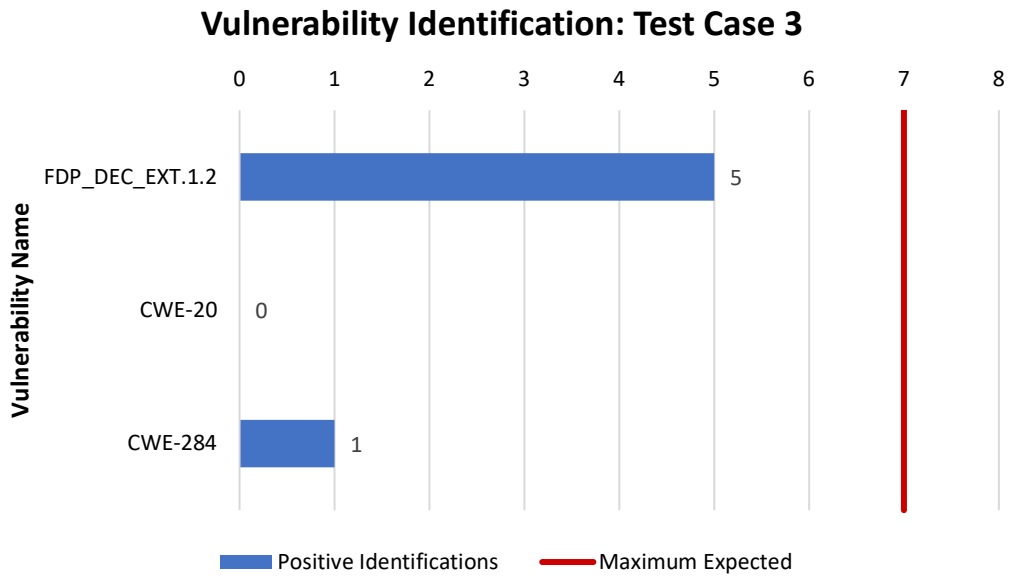

Fig. 5. Vulnerability Identification Test Case 5

Table 20. Test Case 3 Positive Tool Identifications

Vulnerability Name	# Positive Identifications
FDP_DEC_EXT.1.2	5
CWE-20	0
CWE-284	1

4.1.4. Test Case 4

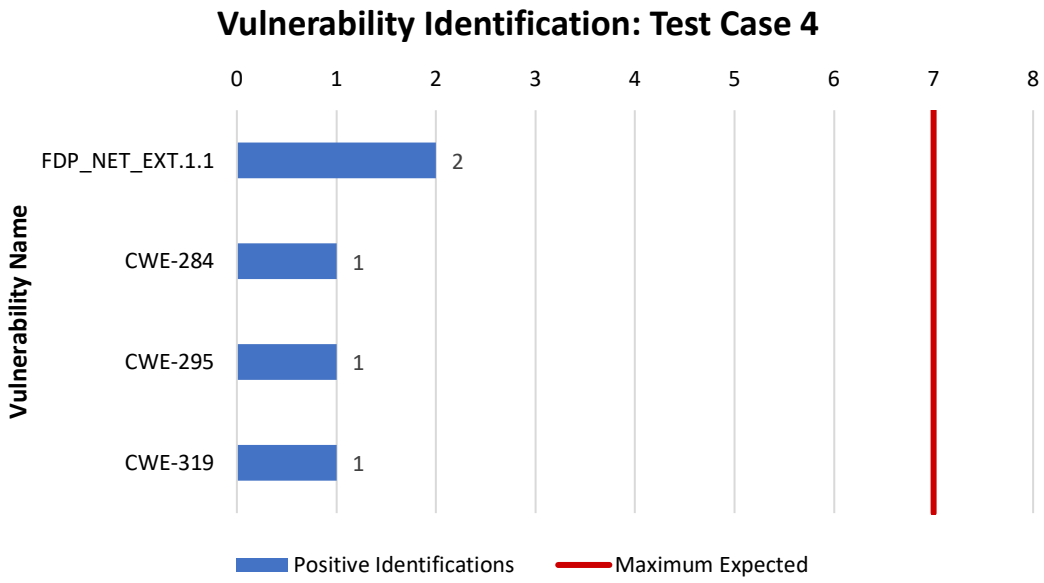

Fig. 6. Vulnerability Identification Test Case 4

Table 21. Test Case 4 Positive Tool Identifications

Vulnerability Name	# Positive Identifications
FDP_NET_EXT.1.1	2
CWE-284	1
CWE-295	1
CWE-319	1

4.1.5. Test Case 5

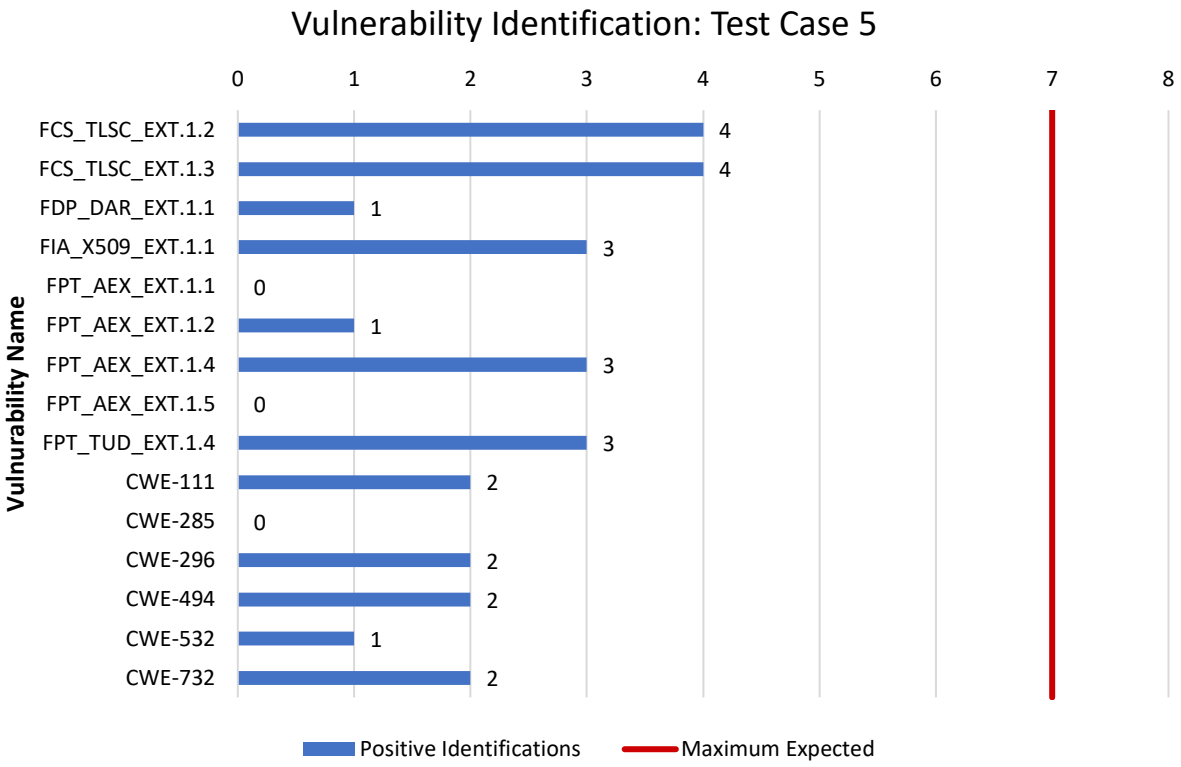

Fig. 7. Vulnerability Identification Test Case 5

Table 22. Test Case 5 Positive Tool Identifications

Vulnerability Name	# Positive Identifications
FCS_TLSC_EXT.1.2	4
FCS_TLSC_EXT.1.3	4
FDP_DAR_EXT.1.1	1
FIA_X509_EXT.1.1	3
FPT_AEX_EXT.1.1	0
FPT_AEX_EXT.1.2	1
FPT_AEX_EXT.1.4	3
FPT_AEX_EXT.1.5	0
FPT_TUD_EXT.1.4	3
CWE-111	2
CWE-285	0
CWE-296	2
CWE-494	2
CWE-532	1
CWE-732	2

4.1.6. Test Case 6

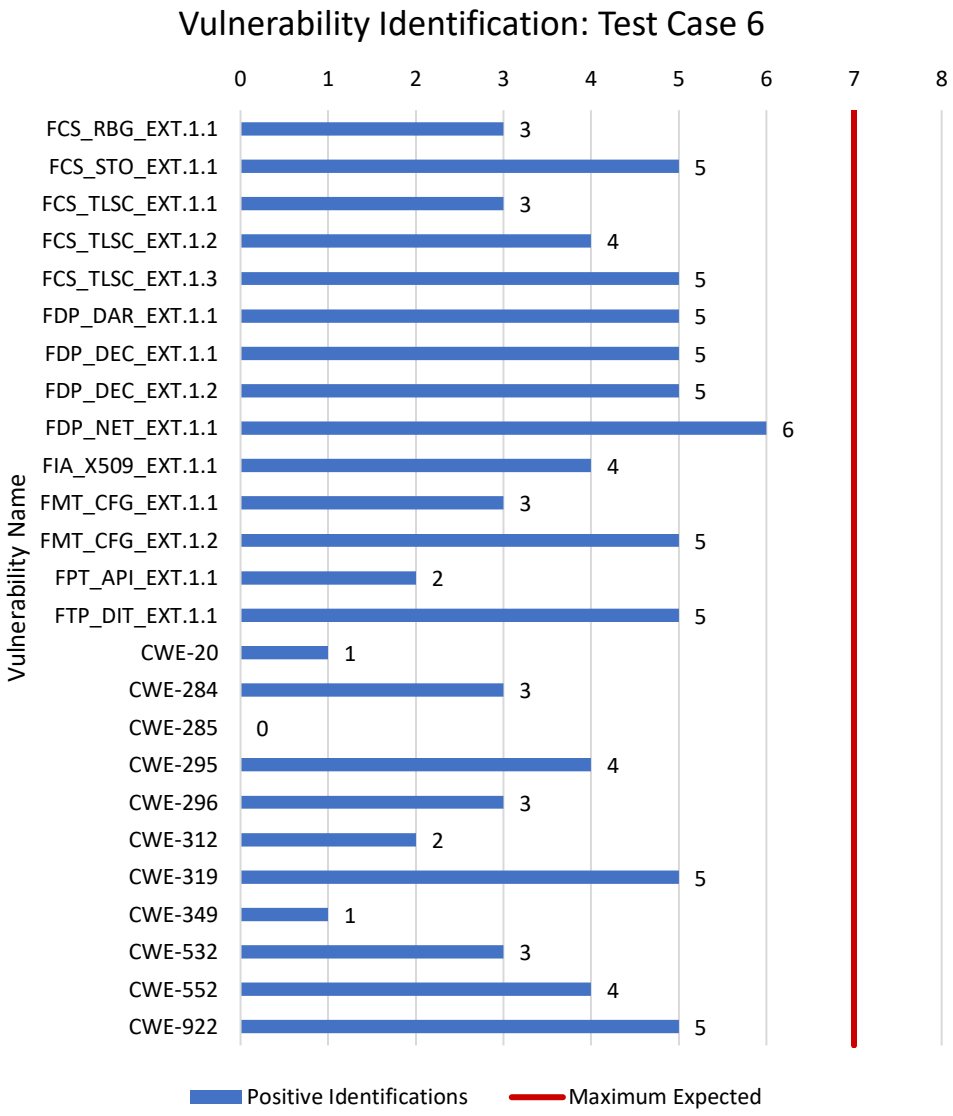

Fig. 8. Vulnerability Identification Test Case 6

Table 23. Test Case 6 Positive Tool Identifications

Vulnerability Name	# Positive Identifications
FCS_RBG_EXT.1.1	3
FCS_STO_EXT.1.1	5
FCS_TLSC_EXT.1.1	3
FCS_TLSC_EXT.1.2	4
FCS_TLSC_EXT.1.3	5
FDP_DAR_EXT.1.1	5

FDP_DEC_EXT.1.1	5
FDP_DEC_EXT.1.2	5
FDP_NET_EXT.1.1	6
FIA_X509_EXT.1.1	4
FMT_CFG_EXT.1.1	3
FMT_CFG_EXT.1.2	5
FPT_API_EXT.1.1	2
FTP_DIT_EXT.1.1	5
CWE-20	1
CWE-284	3
CWE-285	0
CWE-295	4
CWE-296	3
CWE-312	2
CWE-319	5
CWE-349	1
CWE-532	3
CWE-552	4
CWE-922	5

4.1.7. Test Case 7

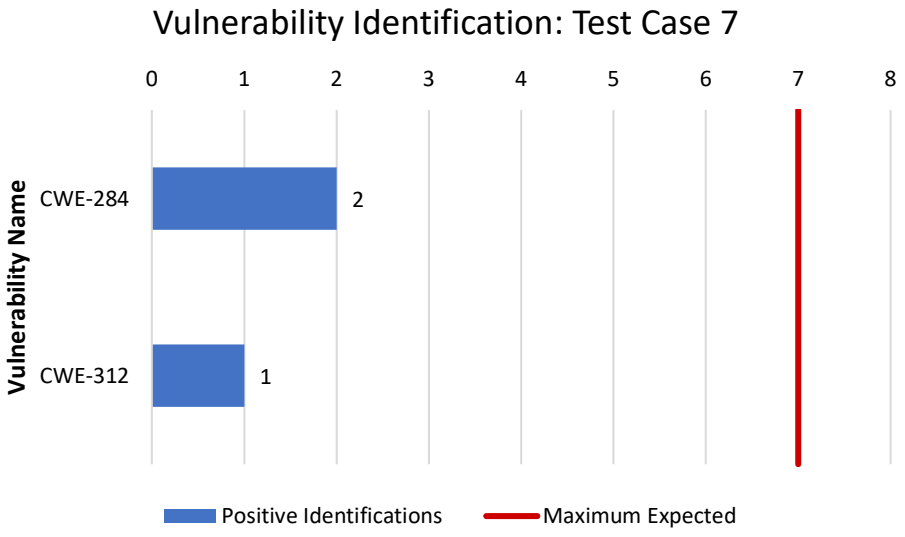

Fig. 9. Vulnerability Identification Test Case 7

Table 24. Test Case 7 Positive Tool Identifications

Vulnerability Name	#Positive Identifications
CWE-284	2
CWE-312	1

4.2. Results

4.2.1. Round Trip Time Per Test Case

The time required for each tool to process each test case was measured in the round-trip time from test case retrieval to test case report submission (see **Fig. 10**).

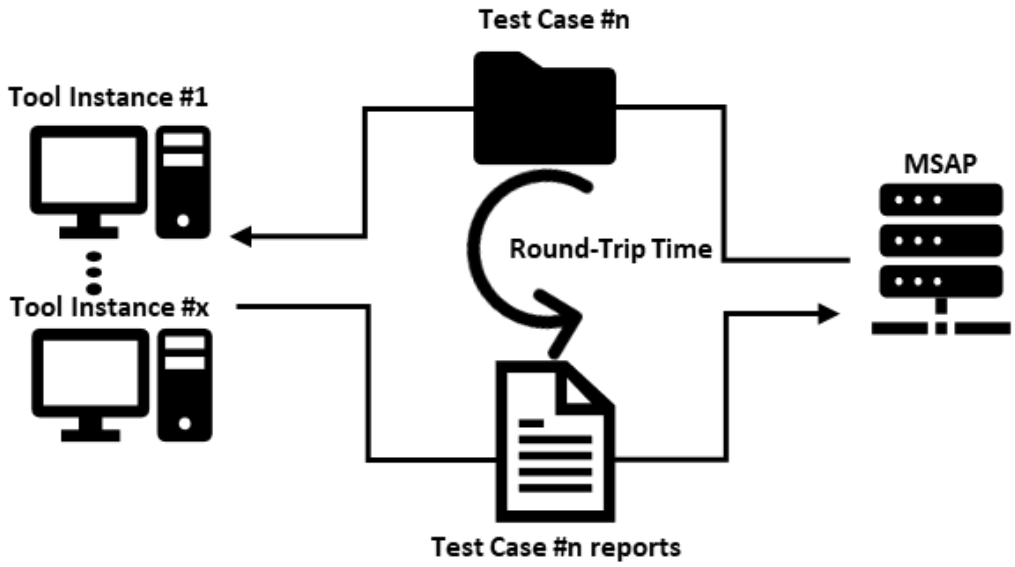

Fig. 10. Round Trip Time

Fig. 11 describes the time taken to complete the analysis of each of the test cases.

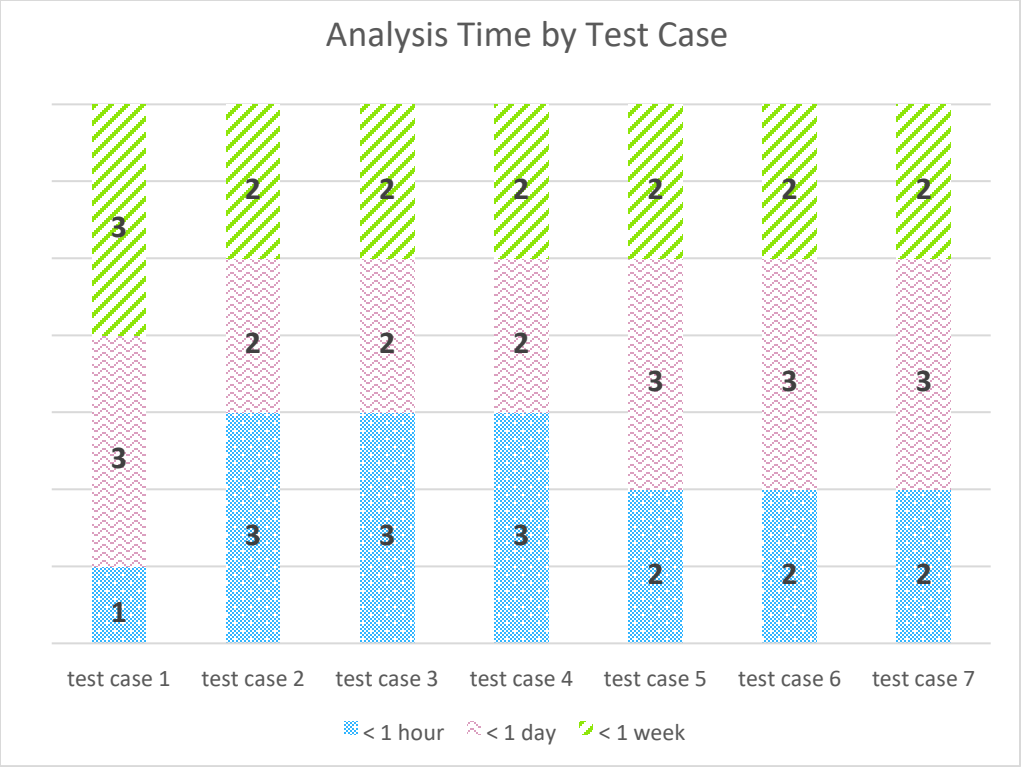

Fig. 11. Analysis Time by Test Case

These times were collected by the orchestration platform described in Section 2.1. For this exercise, the SATE team divided the data set into three buckets:

- analysis that took less than one hour
- analysis that took more than an hour, but less than a day
- analysis that took more than a day, but less than a full week

It should be noted that only seven of the eight total response times are represented in this chart, as the timing information for one of the test sets was deemed inaccurate. Despite this, it can be seen most participants were able to complete their analysis less than one day.

4.2.2. Overall Identification Performance

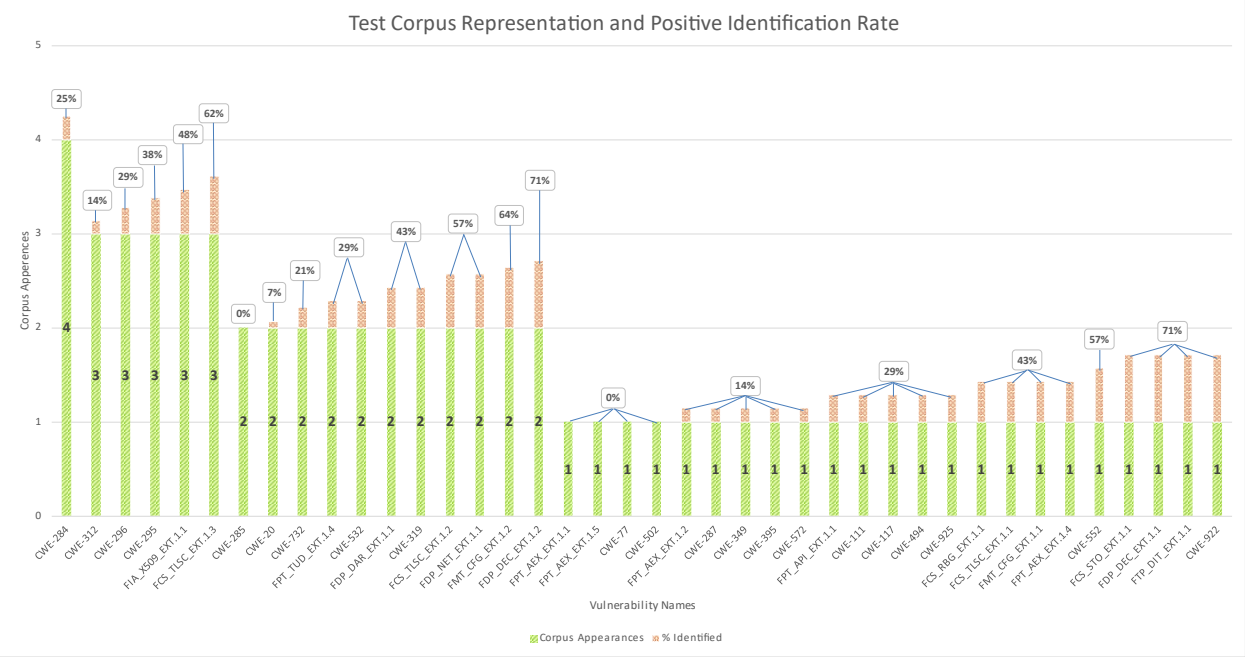

Fig. 12. Misses and Successes Grouped by Vulnerability Name

Fig. 12 shows the number of times a vulnerability (test name) was represented in the test corpus when compared to the positive identification rate for that vulnerability. The test names are ordered first by how frequently they appeared in the test corpus, and then by their positive identification rate. For example, CWE-312 (on the left edge), was represented in three of the seven mobile apps, however, it was only by 14 % of the time by participating tools. For each of the appearance buckets (4,3,2,1) the most frequently identified tests are:

- 4 appearances
  - CWE-284: 25 %
- 3 appearances
  - FCS_TLSC_EXT.1.3: 62 %
- 2 appearances
  - FDP_DEC_EXT.1.2: 71 %
- 1 appearance
  - FSC_STO_EXT.1.1: 71 %
  - FDP_DEC_EXT.1.1: 71 %
  - FTP_DIT_EXT.1.1: 71 %
  - CWE-922: 71 %

Furthermore, five vulnerabilities were not recognized by any tools, in any of the test cases that they appeared in:

- CWE-285
- FPT_AEX_EXT.1.1
- FPT_AEX_EXT.1.5
- CWE-285
- CWE-502

Finally, **Table 25** shows the average positive identification rate for each type of test based on their source (CWE vs NIAP).

Table 25. Average Positive Identification by Test Type

Test Type	# Appearances in Test Corpus	Average Positive Identification
CWE	35	24 %
NIAP	29	45 %
Both	64	34 %

5. Conclusions

5.1. Overall Performance

As was shown in **Table 25**, vulnerabilities were identified at approximately 35 % when looking at all participant groups across all vulnerabilities and all test cases. Furthermore, the identification of CWEs was at roughly half the rate of NIAP vulnerabilities. We believe the positive identification rate would be increased in future efforts with better identifications of CWEs and improved test cases (see Section 5.2). In addition, the exercise shows that tools do indeed find vulnerabilities in real code and a relatively low time investment.

5.2. Lessons Learned

As mentioned in the beginning of this report, Mobile SATE VI is the first instance of a mobile application focused analysis, and the results show there is a lot of space to improve and grow. This section details the primary take-aways from the activity.

5.2.1. Formalized Report Submission Format

To promote ease of participation, the SATE team only placed one requirement on analysis format submission: *mobile app analysis reports needed to simply be a digital report that was readable by the SATE team in some manner*. The goal for this constraint was to enable machine-aided review of the results while not placing unreasonable strain on the participating tools. This

allowed for most tool vendors to provide data in their tool-native formats. However, when the machine-aided review resulted in much lower than expected success rates, we had to fall back on manual evaluation. This resulted in the meta-analysis taking much longer than anticipated and led to delays in meta-analysis phase of the activity. In future iterations of the SATE Mobile Track, a more formalized and structured submission format will greatly improve the response times of this analysis. However, finding the balance between expediated evaluation and added participant burden will be crucial to promote participation.

5.2.2. Mixing Vulnerability Sources

The NIAP and CWE are useful sources for describing vulnerabilities. However, **Table 25** shows that while both types of tests appear roughly with the same frequency throughout the test corpus, NIAP vulnerabilities are identified on average twice as often as CWEs.

We believe this to be more of an artifact of how each of the vulnerabilities is expressed in the application rather than a deficiency in the participating tools. Many of NIAP vulnerabilities used in the test cases are prescriptive and behavioral. They prescribe behavior such as:

> "FIA_X509_EXT.1.1 The application shall [selection: invoked platform-provided functionality, implement functionality] to validate certificates in accordance with … RFC 5280 certificate validation and certificate path validation."

This behavior can be interpolated by examining what ends up resident on the device (binary/artifact files) or by examining the behavior of the application (in this case, altering the certificate and determining if the app rejects network communication. Another example, ***FMT CFG EXT.1.2*** requires all files created by the mobile app, be given restricted permissions to prevent unauthorized access. Detecting a failure in this can "succeed quickly" by simply interrogating the files created by the application during run time[6]. As a contrasting example, the ***CWE-502: Deserialization of Untrusted Data***, requires that a tool can 1) examine source code and 2) introspect deeply into the data flow of the app. Mixing NIAP and CWE vulnerabilities may be unfairly characterizing tools that make no claims to be able to do either.

Future evaluations will need to better describe and cater their test case sets to account for these differences.

5.2.3. Expanding Test Case Resource Pool

Testing the capabilities of mobile app evaluation tools requires an evolving / real-world set of apps with known vulnerabilities to utilize as ground truth. The SATE Mobile Track only had seven test cases available during the evaluation. By contrast, the traditional SATE tracks have tens of thousands of annotated and documented code samples for use during analysis [14]. As part of the exercise, the SATE Team built one of the test cases in house. This represented a significant time/resources investment that will not scale to needs of future exercise. In the future, the SATE team will need to explore new venues to generate these vulnerable apps including the following possible methods:

[6] For the purposes of this argument, the authors are ignoring the difficulty of flexing the function space of an application

- Automated vulnerability injection
- Permuting publicly reported Common Vulnerabilities and Exposures (CVE)
- Targeted crowd-sourcing [15]

References

[1] Ogata MA, Franklin JM, Voas JM, Sritapan V, Quirolgico S (2019) Vetting the Security of Mobile Applications. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-163, Rev. 1. https://doi.org/10.6028/NIST.SP.800-163r1
[2] National Institute of Standards and Technology *Software Assurance Metrics and Tool Evaluation project*. Available at https://www.nist.gov/itl/ssd/software-quality-group/samate.
[3] National Institute of Standards and Technology (2019) *Static Analysis Tool Exposition (SATE) VI*. Available at https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-vi
[4] Howell G, Ogata MA (2017) An Overview of Mobile Application Vetting Services for Public Safety. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 8136. https://doi.org/10.6028/NIST.IR.8136
[5] National Information Assurance Partnership (2016) *Requirements for Vetting Mobile Apps from the Protection Profile for Application Software*. Available at https://www.niap-ccevs.org/MMO/PP/394.R/pp_app_v1.2_table-reqs.htm
[6] The MITRE Corporation (2022) *CWE - Common Weakness Enumeration*. Available at https://cwe.mitre.org
[7] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 1: Introduction and general model. Available at https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf
[8] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 2: Functional security components. Available at https://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R5.pdf
[9] National Information Assurance Partnership (2022) *Approved Protection Profiles*. Available at https://www.niap-ccevs.org/Profile/PP.cfm
[10] Saint-Andre, P. and J. Hodges, Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS) (RFC) 6125. https://doi.org/10.17487/RFC6125
[11] C. Northern, M. Peck (2016) Analyzing the Effectiveness of App Vetting Tools in the Enterprise: Recommendations to the Army Training and Doctrine Command. MITRE Technical Report, MTR160242. (The MITRE Corporation)
[12] VideoLAN Organization (2022) *VLC Media Player.* Available at https://www.videolan.org/vlc/
[13] VideoLabs (2022) *VLC for Android.* Available at: https://play.google.com/store/apps/details?id=org.videolan.vlc
[14] National Institute of Standards and Technology (2022) *Software Assurance Reference Dataset.* Available at https://samate.nist.gov/SARD

[15] National Aeronautics and Space Administration (NASA) (2015) *NASA Uses Crowdsourcing for Open Innovation Contracts.* Available at https://www.nasa.gov/press-release/nasa-uses-crowdsourcing-for-open-innovation-contracts

Appendix A. List of Acronyms

AES
Advanced Encryption Standard

API
Application Programming Interface

CVE
Common Vulnerabilities and Exposures [12]

CWE
Common Weakness Enumeration

GPS
Global Positioning System

IMSI
International Mobile Subscriber Identity

IMEI
International Mobile Equipment Identity

HTTP
Hypertext Transfer Protocol

HTTPS
Hypertext Transfer Protocol Secure

IT
Information Technology

JNI
Java Native Interface

MAC
Message Authentication Code

MSAP
Mobile Security Application Platform

NFC
Near-field Communication

NIAP
National Information Assurance Partnership

OS
Operating System

PP
NIAP Protection Profile

SAMATE
Software Assurance Metrics and Tool Evaluation

SATE
Static Analysis Tool Exposition

SD
Secure Digital

SDK
Software Development Kit

SMS
Short Message Service

URL
Universal Resource Locator

USB
Universal Serial Bus

Appendix B. Glossary

dynamic analysis
Analysis targeting an application's running executable.

protection profile
An implementation-independent set of security requirements for a category of IT products that meet specific consumer needs [17].

result set
The set of mobile app reports associated with a singular analysis tool.

software vulnerability
A security flaw, glitch or weakness found in software that can be exploited by an attacker [1]

static analysis
Analysis targeting an application's source code and/or non-executing binary with the goal of determining unwanted runtime behaviors and characteristics.

test case
In the SATE VI Mobile track, a test case is a mobile application represented by both its compiled binary and its source code.