Detecting Exploitable Vulnerabilities in Android Applications

by

Shivasurya Sankarapandian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

© Shivasurya Sankarapandian 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The world is moving towards remote-first and giving rise to many mobile tools and applications to get the work done. As more applications are moving towards the cloud and therefore require remote access, the attack surface is getting wider. This results in more security vulnerabilities and pain for organizations to manage them. So, organizations have to scale their security operations, and engineers work overtime to detect, verify and mitigate security vulnerability at scale. This includes codebase, infrastructure, corporate assets.For detecting and reporting, security tools are readily available in the market. However, they tend to produce many false-positive results, which are then manually verified by the organization's security engineers. Reproducibility of the security vulnerability and reducing the false positive are the primary goals of the security engineer.

To overcome this challenge, we propose the Detecting Exploitable Vulnerabilities in Android Application framework (DEVAA) to help security engineers to automate security test cases and verify security vulnerabilities at scale. We envision the solution to be incorporated within continuous integration and continuous delivery pipeline.By extending the DEVAA framework similar to JUnit testcase framework, security engineers could automate security testing and verify the actual exploit with feedback from the system without fuzzing them. Additionally, the extension is per vulnerability category type rather than exact vulnerability location which helps security engineers to detect and verify them by leveraging the common framework. DEVAA helps verify security vulnerability flagged by the security scanners by reducing the false positives and confirming security vulnerability reproducibility at scale. Our primary goal while implementing DEVAA is extendability by which security engineers and developers could leverage the base framework to add their application-specific payloads and flows to verify the security vulnerability. Most of the organizations who primarily manage application security and bugbounty programs can leverage DEVAA in implementing well-known security test cases and verifying them in the automated approach.

Acknowledgements

I would like to thank Meiyappan Nagappan for being my advisor and mentor during my Master's. I must also acknowledge other researchers from various publications, this work would not have been possible without their contributions.

Table of Contents

List of Figures viii

List of Tables x

1 Introduction 1

2 Background and Related Work 3

2.1 Background . . . 3

2.1.1 Static Application Security Testing & CodeQL . . . 3

2.1.2 Dynamic Application Security Testing . . . 3

2.1.3 Offensive Penetration testing tools . . . 4

2.2 Related Work . . . 5

2.2.1 Fuzz Testing . . . 5

3 Methodology 6

3.1 Overview . . . 6

3.1.1 Source Code . . . 7

3.1.2 Static Code Analyzer . . . 7

3.1.3 Android Components with vulnerability warnings . . . 11

3.1.4 Vulnerability Runner Component . . . 12

3.1.5 Android Debug Bridge Interface . . . 13

3.1.6 Test-case Driver Component 13
3.1.7 Android Emulator System 13
3.2 DEVAA Workflow 14
3.2.1 Vulnerability Runner and Installation Process 14
3.2.2 Exploit Verification 15
3.2.3 Cross-Site Scripting Attack Overview 16
3.2.4 Content Provider Vulnerability Overview 20
3.2.5 Manual Detection Technique 21
3.2.6 Automated Detection Technique 22
3.2.7 Teardown Test-case 24
3.2.8 Test-case Reporting 24

4 Results **25**
4.1 Initial Experiment 25
4.2 DEVAA with Baseline 26
4.2.1 Open-source Project security vulnerability 26
4.2.2 Cross-Site Scripting on IRCCloud Android Application 26
4.2.3 ContentProvider Attack on VLC Android Application 29

5 Discussion **31**
5.1 Research Questions 31
5.2 Reducing False Positives 31
5.3 Contributions 32
5.4 Threats to Validity 32

6 Conclusions **34**
6.1 Future Work 34

References **35**

A APPENDICES 38

APPENDICES 38

A.1 Our Tools, Artifacts, Results 38

A.2 Figures 38

List of Figures

2.1 DEVAA Value Proposition 4

3.1 Overall Process Overview 7
3.2 CodeQL query example for finding redundant "if" statements in the code . 8
3.3 Content Provider Component Overview 12
3.4 DEVAA Process Overview 15
3.5 Class Diagram for Exploit Driver 16
3.6 Cross-site Scripting Attack in Android WebView Context 17
3.7 Client Server Verification Model 19
3.8 Cross-Site Scripting Attack Verification Architecture Model 20
3.9 Manual Detection of content provider vulnerabilities 22
3.10 Automatic Detection of Content Provider Vulnerabilities 23

4.1 Vulnerable code snippet of IRCCloud Android App - Cross-site scripting vulnerability - com/irccloud/android/activity/ImageViewerActivity.java . . 28
4.2 Vulnerable code snippet of IRCCloud Android App - Cross-site scripting vulnerability - com/irccloud/android/activity/ImageViewerActivity.java . . 29
4.3 Vulnerable code snippet of VLC Android App - Content Provider vulnerability - vlc-android/src/org/videolan/vlc/FileProvider.kt 30

A.1 Screenshot of Web App during Static Code Analysis 39
A.2 Example Security Testcase for IRCCloud Android App - Cross-site scripting vulnerability 40

A.3 Example Security Testcase for VLC Android App - ContentProvider vulnerability 41

List of Tables

3.1 Cross-site scripting source and sink examples in the Android API 10
3.2 source and sink of Exposed Content Provider Vulnerability Android API . 11
4.1 Results of Open-source Android Projects using DEVAA 26
4.2 Comparing Static Code Analysis Results with DEVAA Exploitable Vulnerability Results 27

Chapter 1

Introduction

This thesis focuses on building a generic framework for verifying exploitable security vulnerability in Android apps. Many open-source security scanners and static application security testing tools [7] generate many false-positive results, which require manual verification by the security engineers. However, this does not eliminate the false positive results [5] and they are validated by the security engineers within the organizations. Often, this particular verification, detection process is time consuming and is prone to error as we add manual work to the monotonous verification process. Whereas, our DEVAA framework accepts the results from the static code analysis, processes the information to exploit the issue further, and verifies them in real-time. This includes a framework to drive malicious payloads, create and launch malicious intents and verify the issue post exploiting in the Android app.

Instead of improving the static analysis itself, we suggest to exploit the exact vulnerability automatically. We built a framework which attempts to verify the vulnerability warning at scale and which can be seamlessly integrated within the continuous integration pipelines. Moreover, this particular framework can be adopted by the security engineer and developer to test the code continuously with project specific payloads and detect any abnormal behaviour when the code changes. This common framework is coined as Detecting Exploitable Vulnerabilities in Android Applications (DEVAA) for identifying vulnerabilities in cross-site scripting and content provider categories.

In this thesis, our framework accepts the results from the static code analysis, processes the information to exploit the issue further, and verifies them in real-time. This includes a framework to drive malicious payloads, create and launch malicious intents and verify the issue post exploiting in the Android app. Dynamic Application Security Testing (DAST) [9]

has been widely adopted for finding vulnerabilities in the application by tainting source and sink. However, they are just utilized for identifying vulnerabilities without giving flexible options to execute payloads by the security engineers. Most of the dynamic application security testing tools operate on predefined vulnerable patterns and payloads which which do no offer much customization and extendibility based on static code analysis. As shown in the figure 2.1, DEVAA is built for solving detect and verify vulnerability problem that's overlapping between static analysis, dynamic analysis and offensive pen security testing tools.

The main contributions of this thesis are,

- Building core framework that helps in driving the security test cases.
- Developing test case driver application that can launch crafted intents directly to the vulnerable application in the emulator.
- Building a robust exploit test verification pattern for two example vulnerability types cross-site scripting and content provider attacks.
- Evaluating our framework on three real applications for finding and verifying common cross-site scripting and content provider attacks.

The thesis is divided into following chapters. In chapter 2, relevant literature needed to create this thesis is discussed. In chapter 3, the methodology, process and vulnerability exploits are explained. In chapters 4 and 5, results and implications are discussed. Finally, we conclude our thesis by summarizing our contributions and discussing potential future work.

Chapter 2

Background and Related Work

2.1 Background

Our thesis is an attempt to semi-automate the task of detecting and testing exploitable security vulnerabilities. Though we borrow concepts from unit testing & fuzz testing [4], our thesis's ultimate aim is to confidently help engineers determine the security vulnerability in the source code without dealing with the setup or filtering for false positive cases every time.

2.1.1 Static Application Security Testing & CodeQL

Static Application Security Testing (SAST) [13] is often used by developers and security engineers to find security vulnerability with the well-known vulnerable patterns based on history. These SAST scanner tools helps in generating call-graphs and control-flow graphs. However they don't verify the actual security vulnerability and may contain lot of false positives. CodeQL [10] from Semmle is one of the primary Static Application Security Testing tool which helps to detect common security vulnerability patterns in the source code with predefined query language exposed to the developers, security engineers.

2.1.2 Dynamic Application Security Testing

With rise in more false positive security vulnerability in SAST tools, Dynamic Application Security Testing [9] can be used to detect the security vulnerabilities by scanning and

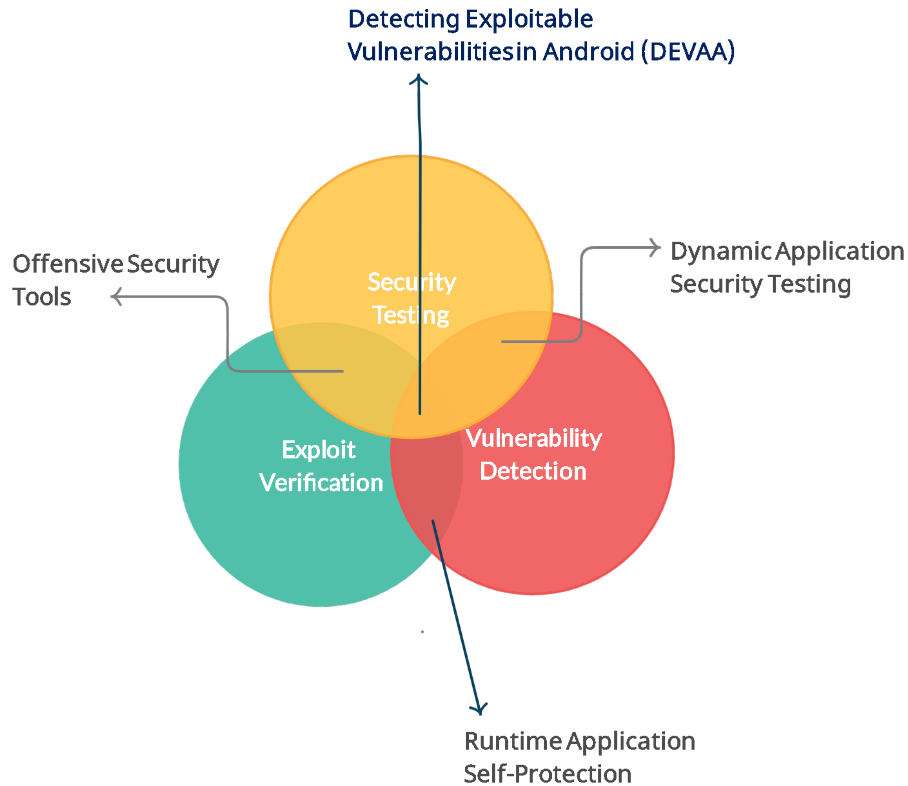

Figure 2.1: DEVAA Value Proposition

analyzing the application at runtime. DAST tools don't necessarily require source code as they rely completely on runtime data to detect malicious attacks. Such an analysis is also called Black-box testing. However, DAST tools don't offer customization and extendability to detect and verify exploits in realtime which is major disadvantage.

2.1.3 Offensive Penetration testing tools

Offensive Penetration testing tools [6] are often utilized by security researchers and hackers to detect, analyze and confirm security vulnerabilities in the codebase. These offensive security tools are capable of exploiting the vulnerability blindly even if the application doesn't have intelligence to infer security vulnerability and considered to be Black-box

testing which is similar to Dynamic Application Security Testing. some of the popular Offensive Security tools are Metasploit [8], NMap [22], HashCat [18] etc.,

2.2 Related Work

Researchers have identified modern way of testing security vulnerabilities in binaries and executable libraries using Fuzz testing helps to help find crash location and provides the log and inputs. Fuzz testing has been successful at discovering security critical bugs in real software [17]. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Fuzz testing verifies security vulnerabilities in Application security layer rather than lower level device drivers.

2.2.1 Fuzz Testing

Fuzz testing has been used for Browser testing [21], Android fuzzing [4] and modern Application security has started leveraging fuzz testing approach for securing the native binaries, applications and mobile application. Vaggelis Atlidakis et al. [3] have come up with fuzzing cloud application and related APIs for properties changes in realtime which leads to security vulnerability. Fuzz tesing methodology can be helpful when the payloads aren't known but instead testing them for diverse set of payload inputs and testing for the behavioural changes in the application. However, these fuzz testing are considered to be black-box testing which doesn't rely on the source code scanning and finding exact vulnerable patterns in the code. The major disadvantage of Fuzz testing is that security vulnerabilities are only found and verified manually whenever a crash occur and requires manual efforts to find the exact memory leak location. However, logical security vulnerabilities which are categorized similar to authorization vulnerabilities, business logic vulnerabilities are not leading to crashes go undetectable with the fuzz testing.

Chapter 3

Methodology

This thesis focuses on building a generic framework for verifying exploitable security vulnerability in Android apps. Many open-source security scanners and static application security testing tools generate many false-positive results, which require manual verification by the security engineers. Whereas our specific framework accepts the results from the static code analysis, processes the information to exploit the issue further, and verifies them in real-time. This includes a framework to drive malicious payloads, create and launch malicious intents and verify the issue post exploiting in the Android app.

3.1 Overview

As shown in Figure 3.1, DEVAA framework integrates various components that work cohesively such as the static code analyzer, Android emulator system, Android Debug Bridge Interface [14]. We chose gradle build system for building the application from the source code which additionally helps static code analyzer to index and search vulnerable patterns in the source code. The results from the static code analyzer are then passed on to the DEVAA framework manually by selecting the Android components with relevant payloads. DEVAA tries establishing connection with the emulator to inject those payloads into the Android application (APK) with the help of crafted intents to execute and verify the results.

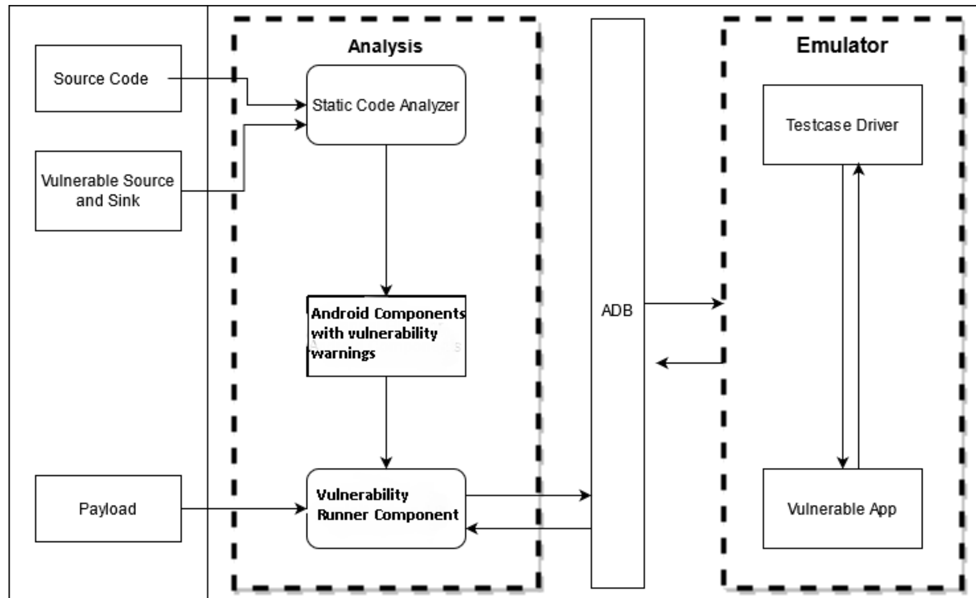

Figure 3.1: Overall Process Overview

3.1.1 Source Code

As a first step, We chose Gradle build system which is default build system adopted by Google, for compiling Android source code. As Java is primarily used for building Android applications, we adopted the gradle plugin for Android java to build, compile and generate application binaries which is helpful for both testing and vulnerable pattern detection process.

3.1.2 Static Code Analyzer

Choosing the right static code analyzer is important as we need to strike the balance between understanding Android build process and Android specific API's while accepting vulnerable patterns as a query. Initially, we tried conducting standard reachability analysis with the help of call graph to map all the vulnerable source and sinks within the code base which created a lot of false positives and had the limitations of traversing across library modules, and annotation processing. Java call graphs are efficient for simple application projects which doesn't resolve dependencies. However, Android application project is more complex in nature which by default utilizes more third-party libraries [11]. Finally, we used CodeQL [10] for code scanning and vulnerability pattern detection, which is a codesearch

```
import java

from IfStmt ifstmt, Block block
where ifstmt.getThen() = block and
  block.getNumStmt() = 0
select ifstmt, "This 'if' statement is redundant."
```

Figure 3.2: CodeQL query example for finding redundant "if" statements in the code

tool developed at Oxford university by the Semmle team and later acquired by Github. CodeQL is able to understand Android build system, index the source code, and provides query interface to interact with the code. Additionally, CodeQL is capable of performing Data flow analysis with the help of implemented methods while traversing the source code from source to sink. Below is a sample CodeQL query 3.2 checks for redundant 'if' statements in the code.

Vulnerable Source and Sink patterns

The primary attack surface of Android application can be Intents [15], File [12], Network and ports [2]. Intents are the primary dispatcher of data between Android components such as activities, providers, receivers and services. Among all the attack surfaces, Intent with exported components shares most of the attack surface area which accept external intents without access control enforcement and that are controlled by the Manifest file AndroidManifest.xml [16]. They could further attack the app remotely by injecting payload into vulnerable network entries in the app.

We primarily use CodeQL for scanning the source code and finding vulnerable patterns within the app. With the help of CodeQL, we could analyze the control flow graph by adding additional conditions globally across the source code while comparing the Call-graph which points only the direct the source and sink method without conditional flows. In this thesis, we have identified vulnerabilities from two categories in the OWASP Mobile

Top 10 [24] as Cross-site scripting (as M1 Improper Platform Usage) and Exposed content provider (as M2 Insecure Data storage attack). With these two vulnerability categories, we have generated the source and sink pattern of the attack to identify the exact flows between the Android components.

Source and Sink for Cross-Site Scripting Vulnerability

Cross-Site Scripting attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites or local pages. Cross-Site Scripting attacks occur when an attacker uses a Intent to send malicious JavaScript code, generally in the form of a browser side script, to a different application. As mentioned in the Table 3.1, the cross-site scripting vulnerability source and sink examples in the Android API which are categorized into three forms.
There are three forms of Cross-Site Scripting, usually targeting users' Mobile Webview:

Reflected Cross-Site Scripting

The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim's browser. Typically the user will need to interact with some malicious link that points to an attacker-controlled page, such as malicious watering hole websites, advertisements, or similar.

Stored Cross-Site Scripting

The application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored Cross-Site Scripting is often considered a high or critical risk. Most source of the stored cross-site scripting are based on developer and project specific as they may be from file, network or database layer. We didnt discussed specifically about stored cross-site scripting in this context as they have wide variety of source and sink pattern.

DOM Cross-Site Scripting

JavaScript frameworks, single-page applications, and APIs that dynamically include attacker controllable data to a page are vulnerable to DOM Cross-Site Scripting. Ideally,

Table 3.1: Cross-site scripting source and sink examples in the Android API

Attack Type	**Source**	**Sink**
DOM cross-site scripting	getIntent#getStringExtras	WebView#loadDataWith BaseURL
Reflected cross-site scripting	getIntent#getStringExtras	WebView#loadUrl
DOM cross-site scripting	getIntent#getDataString	WebView#loadData
Reflected cross-site scripting	getIntent#getDataString	WebView#loadUrl

the application would not send attacker controllable data to unsafe JavaScript APIs. Typical Cross-Site Scripting attacks include session stealing, account takeover, Remote Code execution via Javascript Interface, multi-factor authentication bypass, DOM node replacement or defacement (such as trojan login panels), attacks against the user's browser such as malicious software downloads, key logging, and other client-side attacks.

Source and Sink for Exposed Content Provider

As shown in the figure 3.3 Content providers can help an application manage access to data stored by itself, stored by other apps, and provide a way to share data with other apps. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process. One can easily extend the ContentProvider class and implement their own methods for adding interface for accessing the data source. There is an additional content provider named as FileProvider which is recently added to the development environment to prevent file access abuse and security vulnerability.

Exposed Content Provider which is accessible from the other third party application may have some use-case of sharing data between multiple applications such as authentication and syncing use-cases. However, mis-configurations or bad implementation may cause leakage of sensitive user information from these content providers.

Table 3.2: source and sink of Exposed Content Provider Vulnerability Android API

Attack Type	**Source**	**Sink**
Path Traversal	ContentProvider#openFile	ParcelFileDescriptor#open
ContentProvider mis-configuration	ContentProvider#query	ContentResolver#query
Sensitive File Access	ContentProvider#openFile	ParcelFileDescriptor#open

Content Provider Vulnerability

As shown in the Table 3.2 content provider is one of the basic components in the Android development toolkit which is mapped to resource access directly via special schemes internally within the Android operating system. This implements few methods such as **Query, Update, Delete, Insert and OpenFile** to access raw information directly from the secure sandbox of the application. However, the OpenFile method is commonly abused for the reading sensitive files by utilizing path traversal mechanism.

FileProvider Vulnerability

FileProvider basically extends the ContentProvider class and implements secure access to the internal sandbox files by eliminating known security vulnerability such as Path Traversal issues [25], guessing and brute-forcing file name attacks, symlink attacks.

3.1.3 Android Components with vulnerability warnings

With the generated flows from the CodeQL with the related source and sink in the source code, We try to extract the component information from the results. Additionally, the Android component information is cross-checked with the Manifest file for its existence and access to the component. This component information particularly refers to the Class name or alias, Package, URI schemes, permission information. This extracted component information is represented as JSON by the CodeQL which are then processed manually by the security engineer and check for accessibility from the third party perspective attack surface. This process can be further automated by parsing the JSON information and related mappings to the manifest file.

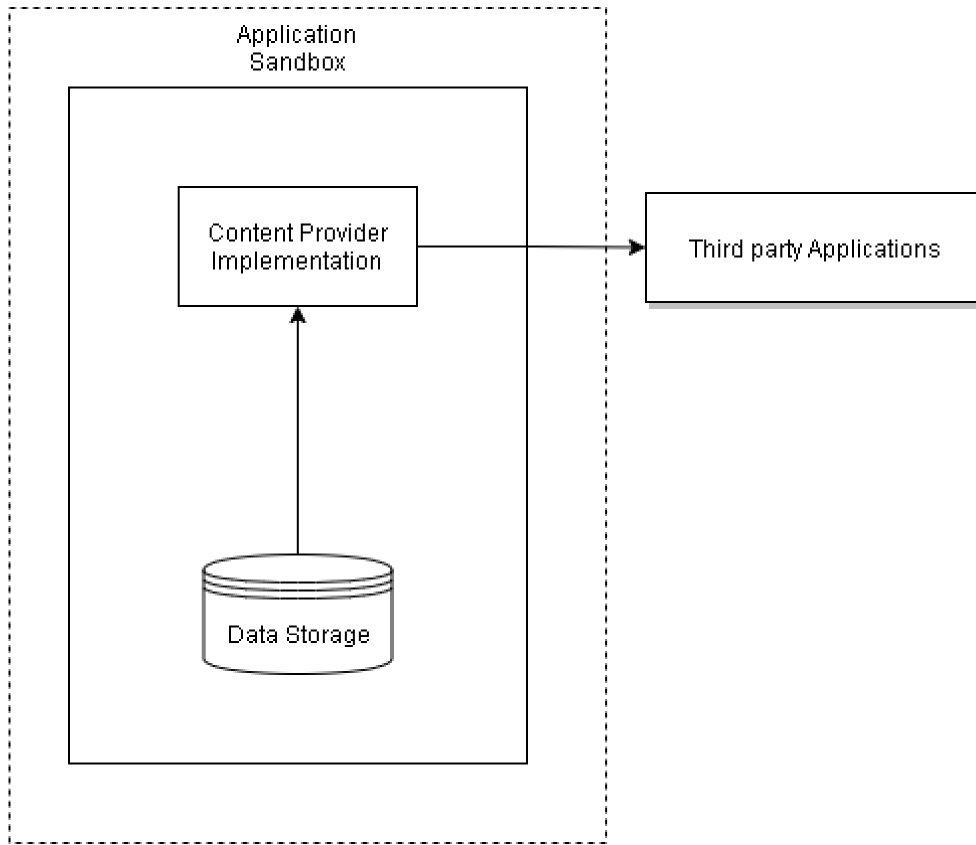

Figure 3.3: Content Provider Component Overview

3.1.4 Vulnerability Runner Component

The vulnerability runner component acts as an bridge between driving test-case and communicating with the Android Debug Bridge. Vulnerability runner components helps security engineers and developers to extend and implement scenarios with payloads for testing the vulnerability to verify the exploit. Similar to Junit Runner, vulnerability runner class can be extended and implemented with runnable methods and provides APIs to communicate with the emulator system. The core capabilities of this component are crafting unique Intent with payloads, specifying Android specific components class, installing Android packages, driving test suite and verifying the exploit.

3.1.5 Android Debug Bridge Interface

Android Debug Bridge Interface which belongs to debugging tools of official Android SDK is the primary communication channel through which interaction between the DEVAA framework and the Emulator system is made. Additionally, exploit verification leverages Android Debug Bridge interface to extract the results from the emulator system and test-case Driver component to verify the exploits.

3.1.6 Test-case Driver Component

Test-case driver component helps the Android Debug Bridge connector to launch the crafted payload to the Android emulator and additionally acts as an proxy for the malicious application. The crafted intent which holds information regarding the payloads, and the components of the vulnerable application are received via Android Debug Bridge and they are converted into native intent before reaching the actual vulnerable application. Additionally, the test-case driver component is helpful in performing data ex-filtration after the attack is successfully executed on the vulnerable application which helps the exploit verifier in the exploit driver component to verify the test-case in realtime. Test-case driver component is an important part in the exploit verification step as they close the loop by extracting the necessary information from the vulnerable application process which is further required for the verification.

3.1.7 Android Emulator System

Android emulator system belongs to official Android development kit and is primarily used for executing all our test-cases against the vulnerable application. Both the vulnerable app and test-case Driver components are executed in a separate sandbox to simulate the exact environment similar to production system. All communication to the emulator system is managed by the Android Debug Bridge interface and shell commands to control the applications.

3.2 DEVAA Workflow

As shown in the figure 3.4, the DEVAA workflow starts with scanning for the known vulnerable patterns in the source code with the help of static code analyzer such as CodeQL. With the help of the warnings from the static code analyzer, we manually extract the Android component information such as class name, intent parameters which is then used by the security engineers to test against different payloads and conditions. Upon successful launch of the crafted intent with the payloads, our test-case driver delivers the payload to the vulnerable application in the emulator system which is then verified by the exploit verifier component by either extracting information from the vulnerable app process or verifying the modification of properties within the application.

3.2.1 Vulnerability Runner and Installation Process

In-order the try exploiting the actual flow returned from the CodeQL, we need to generate a crafted Intent [15] or payload carrier to deliver them from the test driver application to the vulnerable application directly. The vulnerability runner module primarily focuses on accepting payloads, navigating between Android components and initiating the session with the Emulator via Android Debug Bridge interface. Our driver framework acts as base class with number of Android related functions are developed and implemented to facilitate the testing and helping engineers to adopt APIs for developing their own test case scripts for testing the exploits.

```
1 function HTMLEncodingXSSTestCase(package, activity, payload, button_id,
    domain) {
2     initateSession(package, activity)
3     setPayload(payload)
4     clickbyButtonID(button_id)
5     executePayload()
6     verifyScriptingAttack(domain)
7     teardownSession()
8 }
```

Listing 3.1: example exploit driver API

We've added considerable API function support that will be helpful for developers and security engineers to write customized security test case to test the application continuously and generate reports. The class diagram in fig 3.5 gives complete overview of the Exploit driver API and its extendability for developing developer owned security test case scripts.

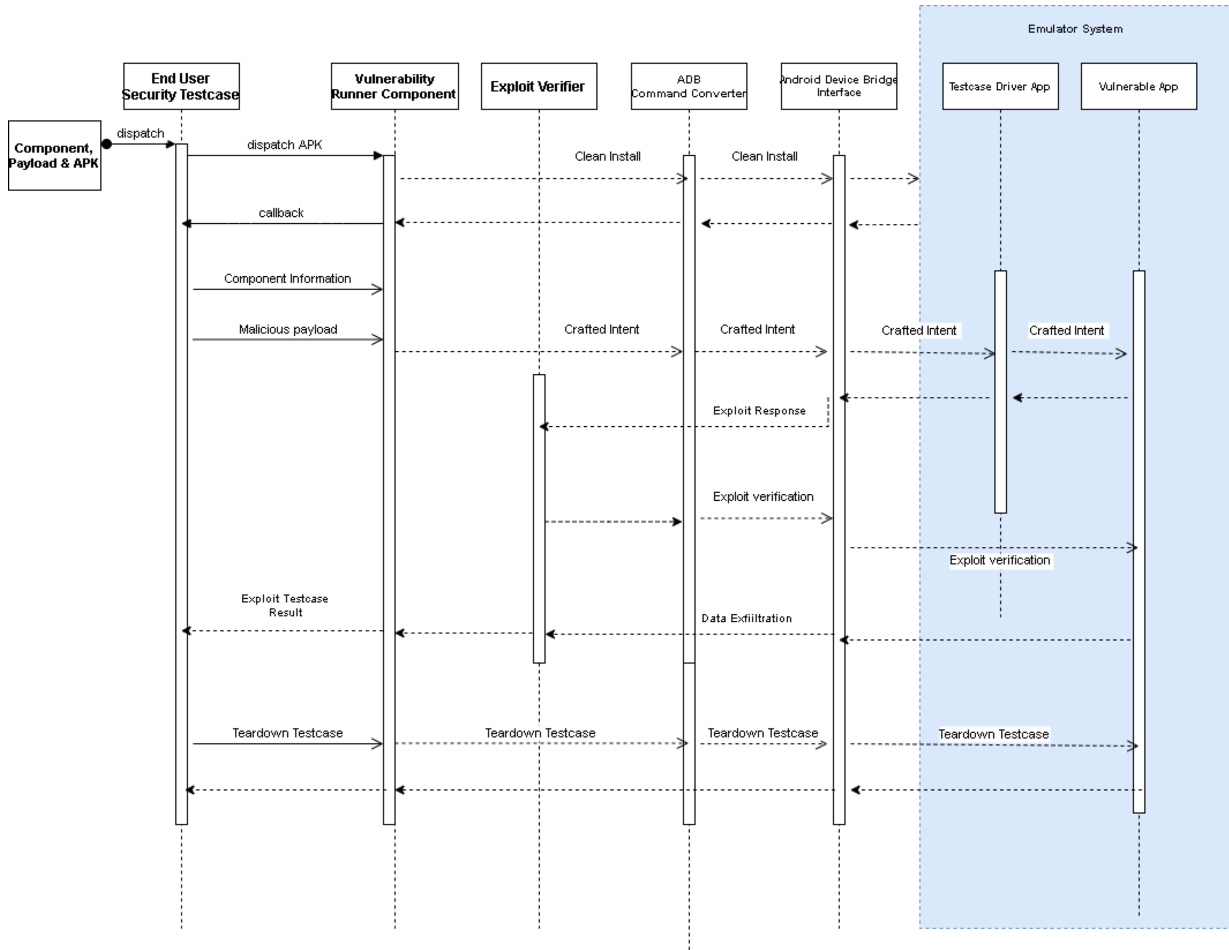

Figure 3.4: DEVAA Process Overview

3.2.2 Exploit Verification

Exploit verification is the core logic of DEVAA framework as it needs to accurately determine the changes in the process and detect for the compromise in the application layer. While comparing it to the fuzzing tools in general, DEVAA primary goal is to verify the security vulnerability exploit-ability from the attack surface and reporting them to the security test-case to reduce the false positive. However in the fuzzing tools, we only check for crashes and not silent exploits. In this section, we discuss about the verification of the vulnerabilities in the application layer by simulating exactly the behaviour of security engineer manually verifying the security vulnerabilities.

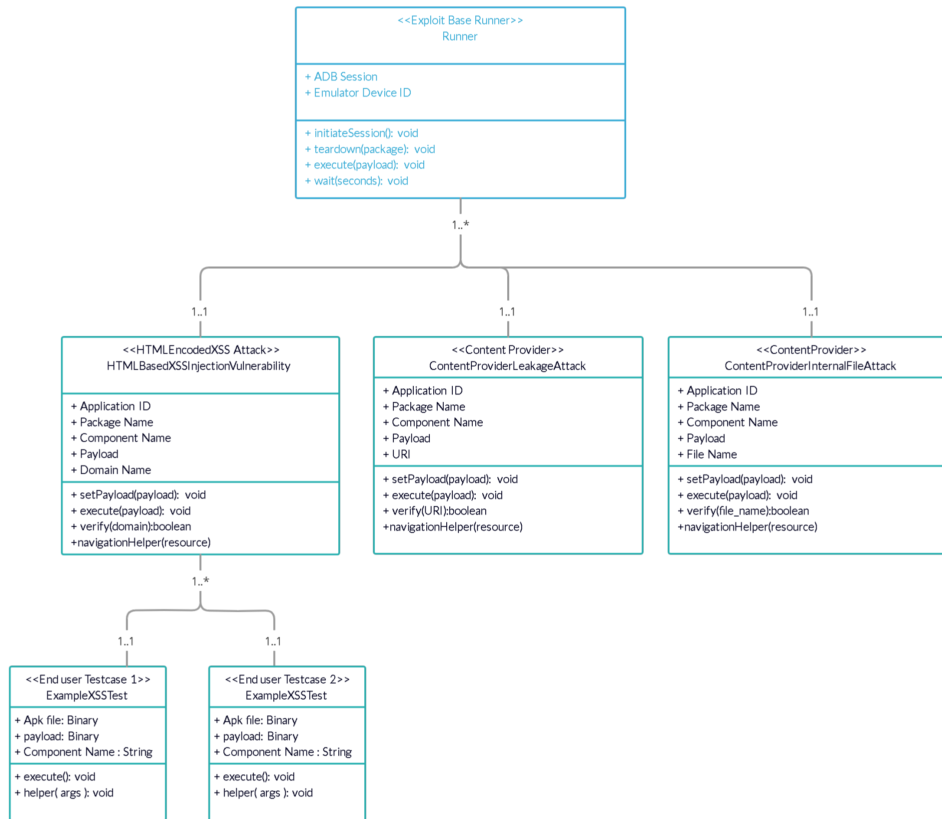

Figure 3.5: Class Diagram for Exploit Driver

3.2.3 Cross-Site Scripting Attack Overview

Cross-site scripting attacks are prevalent and common in the web application context. However, Mobile Webview attack surface are becoming more targeted with a rise in development of Hybrid mobile application using Javascript and HTML5 frameworks. Exposing the whole webview to the Android API makes the attack surface more vulnerable and exploiting the hardware resources such as microphone, and camera. In this subsection, we discuss about the detection of Cross-Site scripting attack and verifying the context of the vulnerability in the Android WebView.

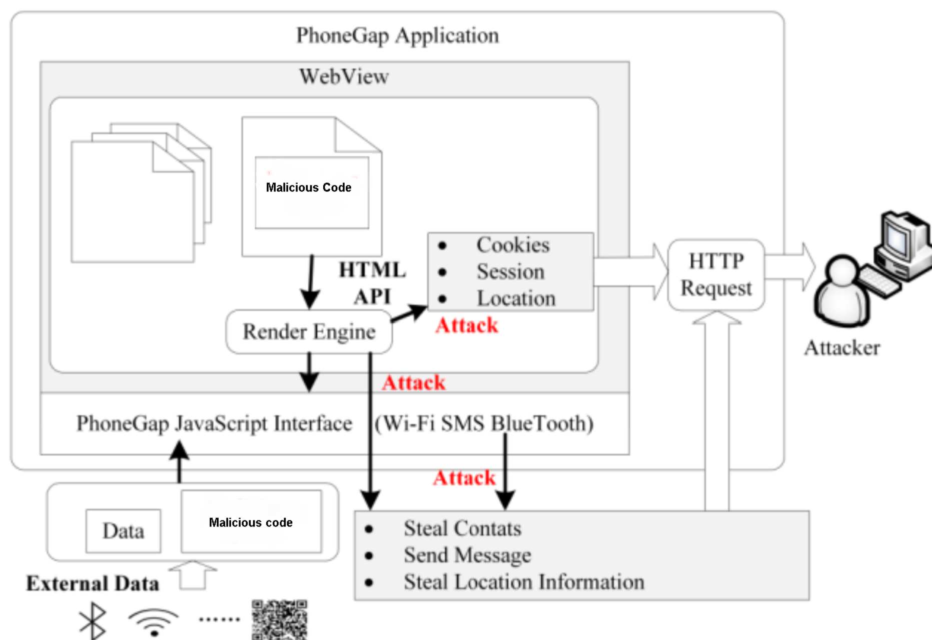

Figure 3.6: Cross-site Scripting Attack in Android WebView Context

Manual Detection Technique

Whenever a cross-site scripting attack is executed in the WebView context, security engineers tend to verify the issue by adding an executable Javascript in the malicious code to verify both the context of the execution framewindow and the result of the execution. The context of the execution framewindow is important as it increases the vulnerability severity as it may hijack user sessions, cookies and information stored within the sandbox environment. The result of the framewindow context and available Android APIs exposed to the WebView are accessed by the WebView console and they are manually verified by the security engineer to determine the severity.

Automatic Detection Technique

In order to automatically verify the cross-site scripting vulnerability, the malicious code injected into the WebView process should permanently change or alter the state of the application within the sandbox. For instance, the cross-site scripting payload can add or modify the cookie [1] state in the webview which is further preserved by the webview

cookie document. This state change within the WebView context should be natural and preserved automatically by the Android framework to prevent false positive cases. These state changes are then automatically verified by the Android Debug Bridge APIs that analyze the result of the attack within the Android application sandbox. Since cross-site scripting rarely extracts or writes to arbitrary internal files in the sandbox, the more generic way is to modify the properties of the WebView which can be further verified even if the application crashes or closes after executing the payload.

Challenges and Failed Attempts of verification

Before coming up with state change technique for the cross-site scripting attack in the WebView, we attempted many techniques to come up with reliable way of verifying the cross-site scripting attack in the Android WebView which eventually failed or had limitations tied to the application specific implementation. In this section, we will be discussing the challenges, approaches and failed attempts with limitation which may be helpful for future optimization and building reliable verification techniques for the cross-site scripting vulnerability categories.

1. Client Server Verification Model

As shown in the figure 3.7, the traditional approach followed in the Web application security model is to attach a command and control script with the malicious script to self detect and execute in the client side which the communicates with the command and control server to extract the information from the client-side and then transmitted to the server-side. The similar approach will be helpful for the Android WebView context to setup and host a command and control server to execute and extract information from the client-side. However, this approach has few major limitations which are listed below

Drawbacks

- Most of the Android Application built using HTML5 framework requires allow-list based host which blocks communication between the compromised client and command & control server
- The Asynchronous verifier script requires co-ordination between the command & control script and exploit verification script that may often lead to timeout in test case based verification setup.

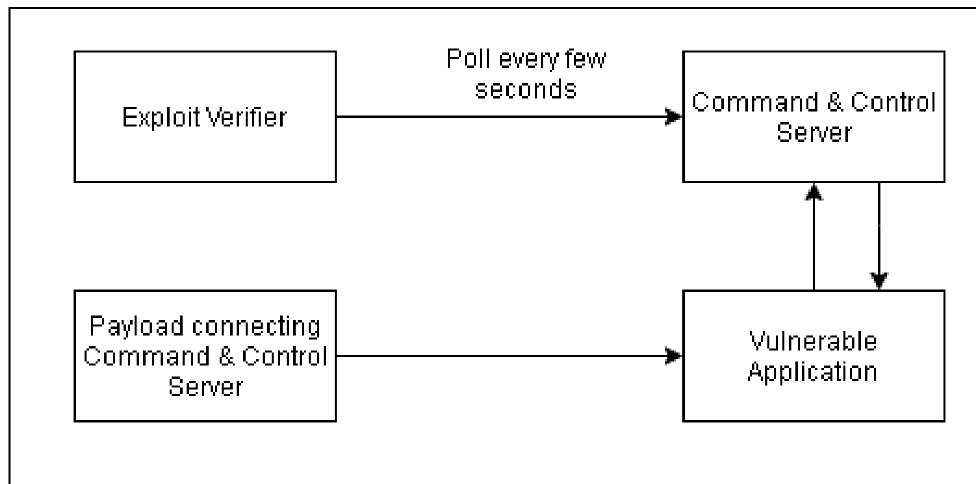

Figure 3.7: Client Server Verification Model

- This client server model requires Internet permission as they are communicated over the network to share the exploit results.

Thus, Client Server verification model has major drawback in terms of verifying the exploit in more generic way instead of targeting the specific application behaviour.

2. Inspecting Application WebView Model

As shown in the figure 3.8, inspecting application WebView from the emulator is a common model used by the security engineers to manually verify the scripting attacks and property changes using debugging tools such developer console, proxy for networking with overall JavaScript execution context. Using the developer console scripting execution engine, the security engineers verify their payload results by either using Logging API calls or making any evidence particularly distinguishing the payload execution results from the trusted script execution context. It is lot more easier to understand the execution context and verify them easily using this inspection method. But the main issues are described below.

Drawbacks

- The hooks for WebView JavaScript execution context aren't publicly available for automating the script detection from the client-side.

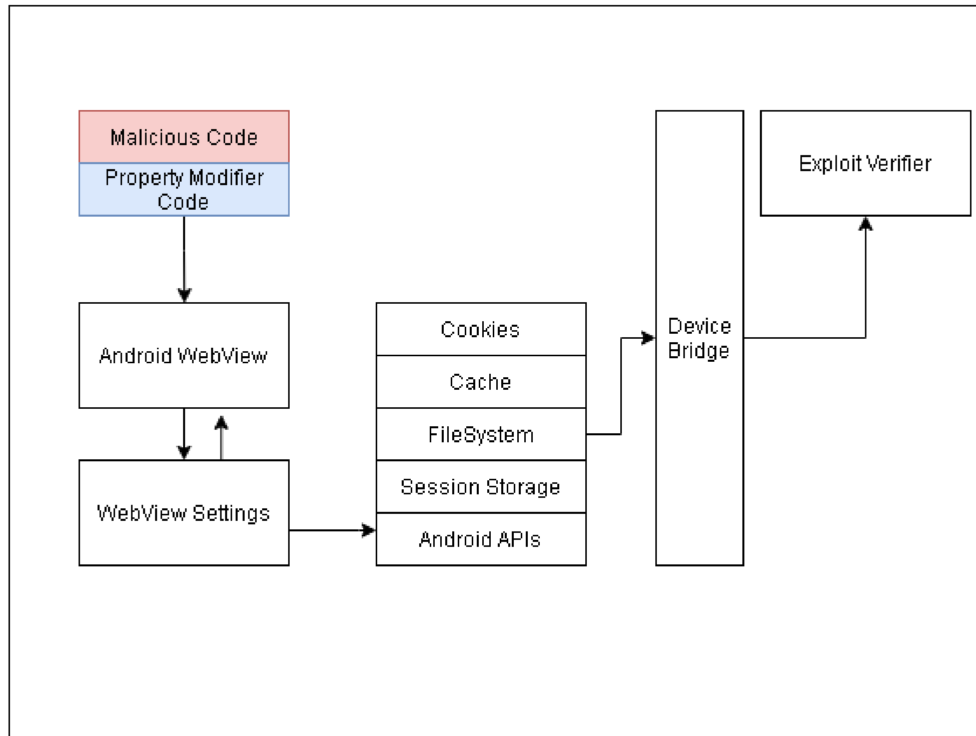

Figure 3.8: Cross-Site Scripting Attack Verification Architecture Model

- Since Android WebView is closely tied with open-source chromium developer tools, the version changes may affect the debugging and detection scripts since the developer tools are modified regularly.
- The lifecycle of the WebView and remote inspecting hook and context should be mapped together which is manually done by the security engineers which is requires lot of heavy-lifting from client-side.

Thus, Application Inspection verification model has successful been used in manual debugging but has major drawback in terms of verifying the exploit in automated manner utilizing the available API's from the client-side.

3.2.4 Content Provider Vulnerability Overview

Content provider is an application level component of Android Development ecosystem which helps to communicate securely between applications from the sandbox environment.

However, this content provider has many attack surfaces due to security mis-configurations, outdated input sanitization techniques. In this section, we will be looking into approaches for detecting exploits and verifying them from the security engineer view-point.

Path Traversal in Content Provider

As per Open Web Application Security Project (OWASP), a path traversal attack (also known as directory traversal) aims to access files and directories that are stored outside the web root folder. By manipulating variables that reference files with "dot-dot-dot-slash (.../)" sequences and its variations or by using absolute file paths, it may be possible to access arbitrary files and directories stored on file system including application source code or configuration and critical system files. It should be noted that access to files is limited by system operational access control (such as in the case of locked or in-use files on the Microsoft Windows operating system).

In Content Provider, file name handling and parsing the uniform resource indicator is so complex. Here are the step by step guidelines followed by the developers to keep the application secure but they fail in most implementation,

- Sanitizing the user inputs by clearing path traversal based special characters.
- Resolving internal sandbox references for the given uniform resource indicator and rejecting them.
- Checking for symbolic reference to the internal sandbox files and rejecting them.
- Checking for access control and permissions before accessing the files

However, most of the implementations are prone to these errors and that creates attack surface over the content provider. In upcoming section, we will discover the process to identify and verify the content provider vulnerability in detailed manner.

3.2.5 Manual Detection Technique

As shown in figure 3.9, content provider vulnerability attack surface is well-defined and has single entry point class which primarily extends ContentProvider or FileProvider and they are primarily exposed to the other applications. So, the payloads are executed directly either from the Provider APIs or Android Debug Bridge APIs by adding vulnerable inputs

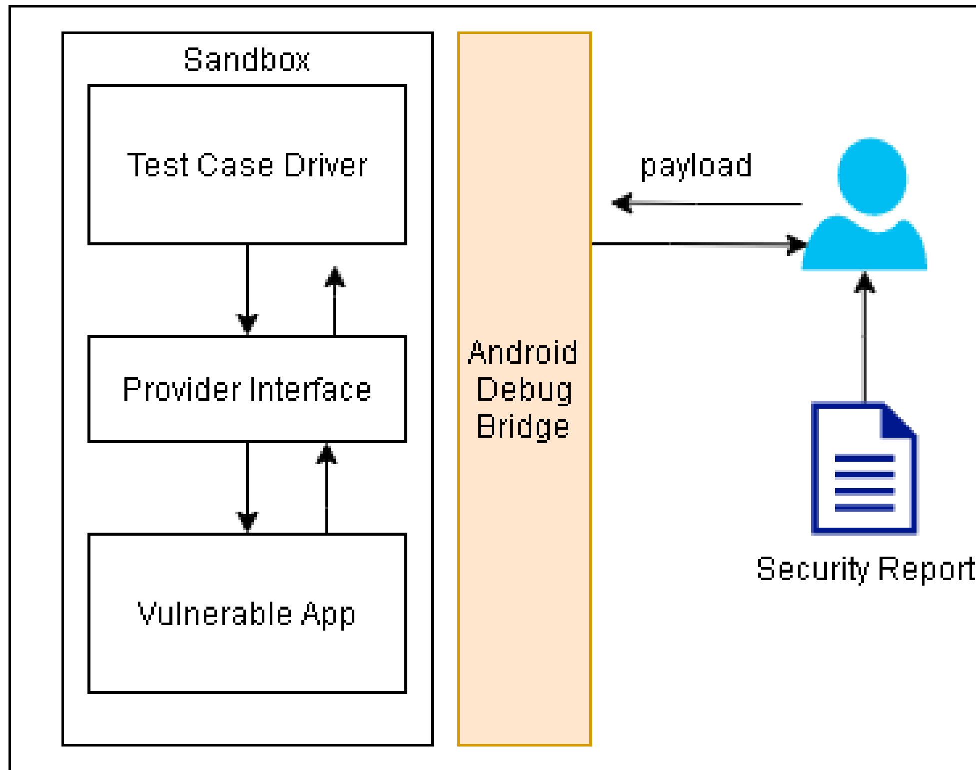

Figure 3.9: Manual Detection of content provider vulnerabilities

and target implementation. As a result, these providers either respond with the leakage of internal sandbox files that may contain sensitive information such as tokens, cookies or overwrite the internal files explicitly. The response format of the providers are often Parcelable file descriptors or binary data which can be easily modified and verified the security engineer to prove the existence of the security vulnerability with the content provider.

3.2.6 Automated Detection Technique

In-order to detect the content provider leakage attacks, the Android API are well documented and stable to retrieve and access them for the verification process. Content provider classes are part of Android Development Kit and Android Debug Bridge supports querying the provider interfaces directly from the command-line which helps security engineers to test and craft the payload directly without interacting with the emulator process. The

process similar to test case where we craft the payload, inject to the process and verify for the result with changes to the process as shown in figure 3.10

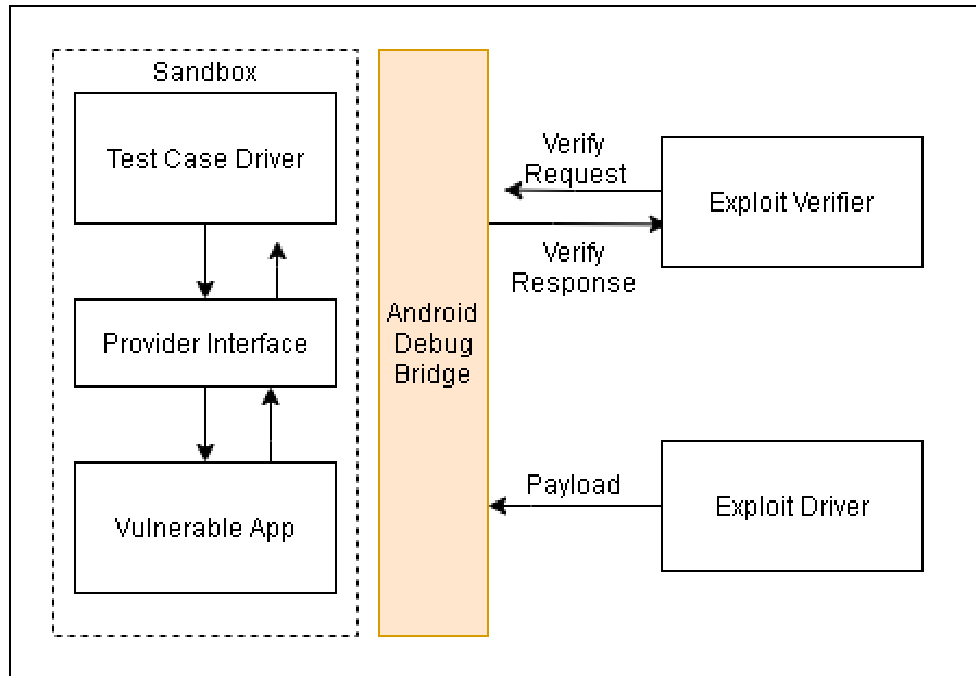

Figure 3.10: Automatic Detection of Content Provider Vulnerabilities

Automated Exploitation Process Steps

Initially, the core test-case framework establishes connection with the Android Debug Bridge process and installs vulnerable application and the test-case driver application separately. Then the core framework accepts the corresponding payload which is specific to the application from the actual security test cases and constructed with the existing framework intent to query the provider interface. The corresponding intents are then passed on to the Android Debug bridge by creating malicious intent to the Test case driver application. The corresponding payload is once again proxied to the vulnerable application by targeting the exact component class. This particular content provider vulnerability is limited to the information leakage only but they do not verify security vulnerability associated with content provider pollution attacks.

Verification Steps

Once the malicious intents are delivered to the application provider component, the verification steps are then triggered from the client-side of the core test-case driver framework. The provider response are then captured by the test-case driver application process as soon the file permission or parcelable file descriptors are exchanged between the vulnerable application and test-case application. Then the parcelable file descriptors and binary information are processed with the help of the vulnerable application process as String. The final results are then passed on to Android Debug Bridge which helps in verifying the file content and exploitability of the reported issue.

3.2.7 Teardown Test-case

After the verification process completes, the result objects are passed as callback to the original test-case caller which helps the actual test-case to verify them with the expected results. Often the results are cross checked by the security engineers with different payloads and test results in action to determine the vulnerability severity, range which may helps in creating Common Vulnerability Scoring System (CVSS) score. Finally, the test-case teardown method is called to uninstall or remove the properties in the emulator setup, clear the Android Debugging session with emulator which facilitates isolation between multiple test-case suite and identify any false positive behaviours.

3.2.8 Test-case Reporting

The final step of the DEVAA workflow is generating test-case report based on the exploit verifier results. Most of the vulnerabilities have different implementation of exploit verification technique as discussed in our previous sections. However, the results are normalized and shared with the actual security test-case with result object containing the actual information and meta data about the vulnerability.

Chapter 4

Results

4.1 Initial Experiment

The initial experiment was learning to build and adopt generic vulnerability call-graph tracing using java call-graph and Soot framework which eventually did not support for adding information such as methods and settings which may lead to true negative cases. We primarily switched to CodeQL from Semmle for extracting vulnerable patterns from source code including the third party libraries source attached. It is important to note that though we use CodeQL for scanning the vulnerability, the patterns that include source and sink with specific criteria are developed and managed by us in-order to reduce false positive results.

Finding security vulnerabilities using patterns and fuzz testing are actively studied by software researcher community. Our main objective is to find reach-ability of vulnerable code from the attack surface and mapping between the vulnerable source & sink with the Android components which is hard to find and novel to experiment with automated techniques. Our initial experiments were with open-source Android projects built using Java and Kotlin which primarily supports gradle build system. CodeQL leverages gradle build system to inject scanning and finding vulnerable patterns when the source code compiles and binaries are generated.

Table 4.1: Results of Open-source Android Projects using DEVAA

security vulnerability	**Source**	**Sink**
Leakage of Internal files exploiting Content Provider - VLC Android	ContentProvider#openFile	ParcelFileDescriptor#open
Directory traversal Attack - VLC Android	ContentProvider#openFile	ParcelFileDescriptor#open
Reflected Cross-Site scripting Attack - IRCCloud	getIntent#getDataString	WebView#loadUrl

4.2 DEVAA with Baseline

This section will showcase results obtained from applying DEVAA to detect exploitable security vulnerabilities on sample vulnerable open-source Android application projects. Results include complete analysis on security vulnerability and code changes.

4.2.1 Open-source Project security vulnerability

Table 4.1 shows the results of finding exploitable vulnerabilities using DEVAA framework. Though the results of Static code analyzer contains few false positive cases, DEVAA has successfully verified the few of the vulnerabilities which is exploitable from the application security layer.

We have identified popular open-source Android projects with active development history and filtered out VLC Player, IRCCloud and Brave Browser Android Application. The primary language for development is Java and Kotlin. VLC Android is an exception as it contains native media libraries written in C++ language and compiled as Linux Shared Object file. Table 4.2 reports the results obtained from Static code analysis and DEVAA test-case results.

4.2.2 Cross-Site Scripting on IRCCloud Android Application

IRCCloud Android app [19] is popular open-source Internet Relay Chat Application Android client written primarily in Java and with few other native libraries. This section

Table 4.2: Comparing Static Code Analysis Results with DEVAA Exploitable Vulnerability Results

security vulnerability	static code analysis vulnerabilities reported	DEVAA vulnerability exploited
Exploiting Content Provider - VLC Android	2	1
Cross-site scripting Attack - VLC Android	0	0
Exploiting Content Provider - IRC-Cloud	0	0
Cross-site scripting Attack - IRC-Cloud	3	1
Exploiting Content Provider - Brave Android	0	0
Cross-site scripting Attack - Brave Android	1	0

contains detailed information about the security report about Cross-site scripting vulnerability in IRCCloud.

Security Report Summary

IRCCloud Android Application uses Android WebView to render images, pastebin URLs, websites in dedicated webview with configuration allowing to execute JavaScript and contains JavaScript Bridge to invoke Android APIs directly from the JavaScript code. However, one of the user input passed on to the webview is directly rendered in the webview without sanitizing for any malicious code or snippets which causes Cross-site scripting attack in the context of IRCCloud Android application webview and able to access the JavaScript bridge.

```java
ImageList.getInstance()
.fetchImageInfo(getIntent().getDataString()
.replace(getResources().getString(R.string.IMAGE_SCHEME), "http"),
new OnImageInfoListener() {
    public void onImageInfo(ImageURLInfo info) {
        if (info == null) {
            ImageViewerActivity.this.fail();
        } else if (info.mp4 != null) {
            ImageViewerActivity.this.loadVideo(info.mp4);
        } else {
            ImageViewerActivity.this.loadImage(info.thumbnail); // by
             ↪  default
        }
    }
});
}
```

Figure 4.1: Vulnerable code snippet of IRCCloud Android App - Cross-site scripting vulnerability - com/irccloud/android/activity/ImageViewerActivity.java

Vulnerable Code Snippet

As seen in lines of 4.1 & 4.2 with the source and sink are getIntent() and loadDataWithBaseURL() respectively, provided the ImageViewerActivity.class was accessible from the external application. This can cause reflected cross-site scripting attack in the context of IRCCloud.

DEVAA security test-case & contribution

DEVAA Security test-case which extends the **HTMLBasedXSS** Attack class accepts generic payload and component information i.e., ImageViewerActivity class name with package name. As mentioned in the methodology section, the exploit driver and verifier drives the payload and verifies the cross-site scripting attack and reports back to the actual security test-case as callback result.

```java
private void loadImage(String urlStr) {
        try {
            // ...
            this.mImage.loadDataWithBaseURL(null, "<!DOCTYPE
             ↪ html>\n<html><head><style>html, body, table { height:
             ↪ 100%; width: 100%; background-color:
             ↪ #000;}</style></head>\n<body>\n<table><tr><td><img
             ↪ src='" + new URL(urlStr).toString() + "' width='100%'
             ↪ onerror='Android.imageFailed()'
             ↪ onclick='Android.imageClicked()'
             ↪ style='background-color:
             ↪ #fff;'/>\n</td></tr></table></body>\n</html>",
             ↪ "text/html", "UTF-8", null);
```

Figure 4.2: Vulnerable code snippet of IRCCloud Android App - Cross-site scripting vulnerability - com/irccloud/android/activity/ImageViewerActivity.java

Mitigation & Severity

The corresponding security vulnerability is fixed in the GitHub source repository [20]. Additionally, the severity of this security vulnerability as determined by CVSS score [23] analysis is 6.9 and classified under generic Cross-site scripting attack.

4.2.3 ContentProvider Attack on VLC Android Application

VLC Player Android app [26] is popular open-source Internet based video player Application Android client written primarily in Java and with few other native libraries. This section contains detailed information about the security report about content provider vulnerability in VLC.

Security Report Summary

VLC Android application has exposed one of the Content Provider interface to share thumbnail images with the third party application especially for Android TV. This provider

```
public ParcelFileDescriptor openFile(Uri uri, String mode)  {
    File file = File(uri.path)
    if (file.exists()) {
        return ParcelFileDescriptor.open(file,
         ↪  ParcelFileDescriptor.MODE_READ_ONLY)
    }
    throw FileNotFoundException(uri.path)
}
```

Figure 4.3: Vulnerable code snippet of VLC Android App - Content Provider vulnerability - vlc-android/src/org/videolan/vlc/FileProvider.kt

is vulnerable to both directory traversal attack and internal file access attack which leads to leak sandbox files and tokens without end-user knowledge.

Vulnerable Code Snippet

As seen in Lines in 1 and 4 of 4.3 with the source and sink are openFile() and ParcelFileDescriptor#open() provided the FileProvider.class was accessible from the external application caused internal file data leakage in the context of VLC Android application.

DEVAA security test-case & contribution

DEVAA Security test-case which extends the **FileProviderRunner** Attack class accepts generic payload and component information i.e., FileProvider class name with package name. As mentioned in the methodology section, the exploit driver and verifier drives the payload and verifies the content provider data leakage attack and reports back to the actual security test-case as callback result.

Mitigation & Severity

The corresponding security vulnerability is fixed in the GitHub source repository [27]. Additionally, the severity of this security vulnerability as determined by CVSS score [23] analysis is 8.0 and classified under generic path traversal and unintended data leakage attack.

Chapter 5

Discussion

5.1 Research Questions

RQ1: *Can DEVAA be successfully applied to exploit any generic Android Application?* **Yes.** With the access to compilable source code and all dependency resolved, we could apply vulnerability detection pattern to find vulnerabilities and try exploiting with the defined payloads with the corresponding vulnerable Android application developed using Java and Kotlin language. As Exploit driver code can be extendable and accepts payload with component information, We could add generic Android application for testing as long as a source and sinks are clearly defined.

RQ2: *Can DEVAA successfully work with other static code analyzer tools apart CodeQL?* **Yes.** As of now DEVAA relies on results CodeQL but the vulnerability patterns including source, sink are added as input are maintained within the project. As far as any Static Code Analyzer that can produce output containing vulnerable pattern location & component information, We could successfully extract and use it in our exploit module directly.

5.2 Reducing False Positives

Although the ultimate aim of DEVAA is to reduce false positive, there might be cases where the Android specific components aren't exposed to the third party application as mentioned in the Android manifest file. They are still potential security vulnerability

without actual exploit and marked as safe but they are still a recommendation for the developers and security engineers to fix the reported security bug.

Another major shortcoming was to tweak Static Code Analyzer by adding more source and sink with number of pass through configuration checks. In ideal scenario, Developers tend to use lot of sanitization function before even invoking the vulnerable Android sink API's. Static code analyzer can still flag them as potential security vulnerability but DEVAA would mark them safe if the code flow and exploit-ability of security vulnerability isn't reachable. In such cases, tweaking the static code analyzer to ensure the standard sanitizing functions is more appropriate way to find even more security vulnerability and bypasses around the code.

5.3 Contributions

We have presented the contributions of our study as follows:

- We have provided couple of vulnerability category namely cross-site scripting and content Provider data access with exploitation scripts, sample code snippets to test drive the security test-case.
- By demonstrating DEVAA on two distinct vulnerability that it is possible to build and test generic security test-case suite to test different vulnerability category with Android applications.
- We have open-sourced all the static code analysis processing interface, vulnerability pattern for cross-site scripting attack, content provider data access and exploit verification scripts.
- We have found and verified couple of previous security vulnerabilities with IRCCloud and VLC Android application with the help of DEVAA.

5.4 Threats to Validity

- DEVAA relies on CodeQL static code analysis tool for processing and finding vulnerability patterns and they may not reflect the exact pattern matching or call-graph based on our search query and result in true negative results in the source code.

- Diversifying the vulnerability pattern for all configuration based on developer usage or pattern is ideal to find security vulnerability in generic manner. As the codebase and DEVAA evolves the diversification is possible and this may result in true negative results.

- Reproducing the exact vulnerability may require additional setups within the Android application such as signup or login which is usecase specific driven and current DEVAA tool may lack in few navigation API options.

- Reproducing the exact security vulnerability may be challenging with different emulators and factors including CPU architecture, Android API levels, Development kit updates. These selections are often made by the security engineer manually while testing the application.

- We only derived generic exploit verification technique for cross-site scripting and content provider issues covering few Android vulnerable APIs. As discussed before, diversifying the pattern and payload is highly recommended for security testing to reduce false positives and bringing up true negatives.

- Currently our framework can scan and operate on Java & Kotlin based compilable gradle supported Android studio projects and limitation on legacy Android projects. Hybrid apps that are powered by Java are still available to scan for security vulnerability but the primary JavaScript language isn't supported with DEVAA.

Chapter 6

Conclusions

In our thesis, we have built a generic and extendable security testing tool that can help security engineers and developers to automate security testing and help in finding exploitable vulnerabilities at scale. We have demonstrated the security tool with common security vulnerabilities on real-life projects. We hope that this would encourage more security researchers and engineers to adopt, extend and contribute to build a payload and vulnerability corpus that may help to detect exploitable vulnerabilities. Our work was heavily influenced by the creating developer tools for finding reproducible security bugs and day-to-day life of security engineers time to verify the security vulnerabilitys in larger organizations putting the entire stack in risk.

6.1 Future Work

- Adding more payloads and vulnerability categories to help Mobile Application developers, security engineers in testing.
- Building Android Studio plugin for developers to automate security testing in few clicks without leaving the development environment.
- Adding support for other programming language excluding Java and Kotlin which includes React native and hybrid mobile apps which is getting popular over time.
- Building a corpus of pre-defined security test-cases and common vulnerabilities with payloads that may help in finding and scanning source code for Android studio projects.

References

[1] Cookies: Overview. https://en.wikipedia.org/wiki/HTTP_cookie, 2017.

[2] Researchers: Open android ports leave millions vulnerable. https://adtmag.com/articles/2017/05/03/open-port-security.aspx, 2017.

[3] V. Atlidakis, P. Godefroid, and M. Polishchuk. Checking security properties of cloud service rest apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397, Oct 2020.

[4] Android Documentation. Android fuzzing project. https://source.android.com/devices/tech/debug/libfuzzer, 2017.

[5] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008).

[6] Software Testing Help. Offensive security tools. https://www.softwaretestinghelp.com/penetration-testing-tools/, 2020.

[7] Melina Kulenovic and Dzenana Donko. A survey of static code analysis methods for security vulnerabilities detection. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1381–1386, 2014.

[8] Metasploit. Metasploit tools. https://www.metasploit.com/, 2020.

[9] Microfocus. Dynamic application security testing. https://www.microfocus.com/en-us/what-is/dast, 2020.

[10] O. Moor, D. Sereni, M. Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and J. Tibble. .ql: Object-oriented queries made easy. In *GTTSE*, 2007.

[11] Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. Understanding reuse in the android market. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122, 2012.

[12] Check Point Software. Man in the disk attack surface. https://blog.checkpoint.com/2018/08/12/man-in-the-disk-a-new-attack-surface-for-android-apps/, 2018.

[13] Synopsys. Static application security testing. https://www.synopsys.com/glossary/what-is-sast.html, 2020.

[14] Android Team. Android debug bridge tool. https://developer.android.com/studio/command-line/adb, 2020.

[15] Android Team. Android intent framework. https://developer.android.com/guide/components/intents-filters, 2020.

[16] Android Opensource Team. Android manifest file content. https://developer.android.com/guide/topics/manifest/manifest-intro.

[17] Google Security Team. Project zero. https://googleprojectzero.blogspot.com/, 2020.

[18] HashCat Opensource Team. Hashcat tools. https://hashcat.net/hashcat/, 2020.

[19] IRCCloud Development Team. Irccloud android application. https://play.google.com/store/apps/details?id=com.irccloud.android&hl=en_CA&gl=US.

[20] IRCCloud Security Team. Irccloud security issue fix commit. https://github.com/irccloud/android/commit/8ff145519bcd30da1898dd54a68629f53c62afe7.

[21] Mozilla Opensource Team. Firefox browser fuzzing project. https://hacks.mozilla.org/2021/02/browser-fuzzing-at-mozilla/, 2020.

[22] NMap Team. Nmap tools. https://nmap.org/, 2020.

[23] NVD NIST Team. Cvss score metric. https://nvd.nist.gov/vuln-metrics/cvss.

[24] OWASP Team. Owasp mobile top 10 security vulnerabilities. https://owasp.org/www-project-mobile-top-10/, 2020.

[25] OWASP Team. Owasp path traversal vulnerability. https://owasp.org/www-community/attacks/Path_Traversal, 2020.

[26] VLC Development Team. Vlc android application. https://play.google.com/store/apps/details?id=org.videolan.vlc&hl=en_CA&gl=US.

[27] VLC Security Team. Vlc security issue fix commit. https://code.videolan.org/videolan/vlc-android/commit/86051dd9753a126e454726d9141566d4b1999262.

Appendix A

APPENDICES

A.1 Our Tools, Artifacts, Results

Links to our code, tools & and results are as follows:

- GitHub link to DEVAA Project
- TestCase Driver Android Application
- Nextcloud Android Source
- IRCCloud Android Source
- CodeQL Reference Example

A.2 Figures

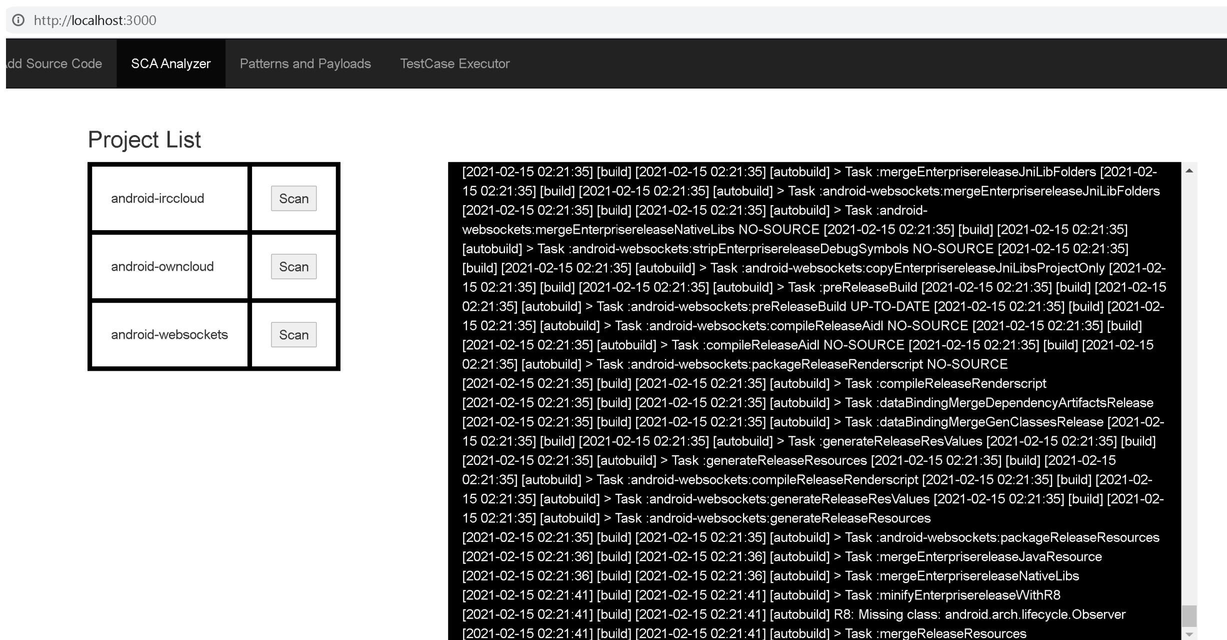

Figure A.1: Screenshot of Web App during Static Code Analysis

```
let HtmlEncodedXSS = require("./HtmlEncodedXSS");
class IRCCloudXSSTestCase extends HtmlEncodedXSS {
    constructor(name) {
        super();
        this.xssPayload =
        [
            "https://picsum.photos/500",
            "https:/jbdaksndf.com/dskjhbakjhsfh",
        ];
        this.domain = [
            "https://zoho.com/",
            "twitter.com"
        ]
        this.eventbasedXSS();
    }

    eventbasedXSS() {
        this.addPayloads(this.xssPayload[0], "com.irccloud.android",
         ↪  "com.irccloud.android.activity.ImageViewerActivity",
         ↪  this.domain[0]);
        this.executePayloads();
        this.clearActivity();
        this.clear();
        this.assertTrue("/data/data/com.irccloud.android
        /app_webview/Cookies", "zoho.com");
    }
}
var IRCCloudXSSTestCase1 = new IRCCloudXSSTestCase();
```

Figure A.2: Example Security Testcase for IRCCloud Android App - Cross-site scripting vulnerability

```
let FileProviderRunner = require("./FileProvider");
class VLCFileProviderTestCase extends FileProviderRunner {
    constructor(name) {
        super();
        this.VLCPayloads =
        [
            "content://org.videolan.vlc.thumbprovider/data
            /data/org.videolan.vlc/databases/vlc_database",
        ];
        this.directoryTraversalTestCase1();
        this.sandboxBypassTestCase2();
    }

    directoryTraversalTestCase1() {
        this.clear();
        this.addPayloads(this.VLCPayloads[0]);
        this.executePayloads();
        this.assertTrue("exfiltrated-shared-pref-mei.xml");
        this.clear();
        this.removeTemporaryFiles();
    }

    sandboxBypassTestCase2() {
        this.clear();
        this.addPayloads(this.VLCPayloads[0]); // sets the payload
        this.executePayloads(); // executes the payload
        this.assertTrue("exfiltrated.sqlite");
        this.clear(); // teardown testcase suite
        this.removeTemporaryFiles();
    }
}
var VLCFileProviderTestCase1 = new VLCFileProviderTestCase();
```

Figure A.3: Example Security Testcase for VLC Android App - ContentProvider vulnerability