USENIX Security '24 Artifact Appendix: DVa: Extracting Victims and Abuse Vectors from Android Accessibility Malware

Haichuan Xu[1], Mingxuan Yao[1], Runze Zhang[1], Mohamed Moustafa Dawoud[2], Jeman Park[3], Brendan Saltaformaggio[1]
[1]*Georgia Institute of Technology* [2]*German International University* [3]*Kyung Hee University*

A Artifact Appendix

A.1 Abstract

The artifact is a code repository (with supporting documentation) for DVa, an automated symbolic execution pipeline used to perform analysis of Android accessibility (a11y) malware. DVa's major component is the static symbolic execution of a malware APK that outputs the malware's targeting victims, abuse vectors, and persistent mechanisms. DVa also consists of peripheral dynamic hooking and analysis code that combats malware's anti-analysis techniques to aid the loading of malicious payloads.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Due to the unique nature of malware analysis, DVa is designed to interact with malware actively and its potentially live C&C servers while revealing user-side environmental and victim app information to the malware. This is necessary to combat malware's anti-analysis techniques such as packing, dynamic code loading, and victim app scanning to aid in revealing malware's abuse payloads. However, for the artifact evaluation, we have prepared the malware sample in a way that should not pose any inherent security, privacy, or ethical concerns while obtaining the major claims of the paper. In specific, no analysis is performed to transmit data with malware's live C&C servers. If the user decides to run active malware to verify DVa's claims, they do so at their own risk.

A.2.2 How to access

The artifact is a stable reference to a tree with a documented tutorial that can be accessed on GitHub: https://github.com/CyFI-Lab-Public/DVa/tree/623337245d2588a6b87bc9bb7791497c4251d787. We also included a compiled executable JAR file with dependencies to ease the setup of static symbolic execution.

A.2.3 Hardware dependencies

DVa's static symbolic execution requires a Linux machine. The framework was tested on a Ubuntu 20.04.6 LTS machine with a 24-core CPU and 128GB Memory. However, DVa should work with any recent version of Ubuntu or any Debian 11 and up (64-bit) machines.

DVa's peripheral dynamic analysis and hooking framework requires physical Google Pixel 3 phones (with OEM unlock). To maximize the chance of creating a valid execution environment of malware, DVa uses five physical Google Pixel 3 phones to execute the dynamic analysis. The phones run patched version of Android 9 blueline images with version image-blueline-pq1a.181105.017.a1.

A.2.4 Software dependencies

DVa's symbolic execution is built on top of *Soot*. Software dependency for running the compiled JAR version of DVa is the OpenJDK version 1.8.0_412 with Java Runtime Environment.

DVa relies on *adb*, *python 3.10.12*, *Magisk*, and *EdXposed* to run the dynamic hooking framework. For details, please refer to the ROM flash and dynamic hooking framework installation instructions in the GitHub repository.

A.2.5 Benchmarks

The primary benchmark used in the paper is a collection of Android a11y malware (malware that requests the `BIND_ACCESSIBILITY_SERVICE` permission) collected from VirusTotal Intelligence. This dataset is used to examine DVa's effectiveness in detecting a11y malware's targeting victims, abuse vectors, and persistence mechanisms. The benchmark was run on a Ubuntu 20.04.6 LTS system with DVa deployed. The results are shown in Tables 5 and 6 of the paper.

A.3 Set-up

A.3.1 Installation

Users should follow the Setup section of the README to deploy DVa. Install Java 1.8 by running `sudo apt install openjdk-8-jdk`.

A.3.2 Basic Test

The Usage section of the README contains a step-by-step instruction to run DVa against malware *Pixstealer*, an Android a11y malware that eavesdrops on user credentials and conducts automated unauthorized transactions abusing banking applications. In short, to use the compiled JAR with all dependencies packaged, the user should:

1. `cd` to DVa's directory.
2. Run DVa's malware analysis module against the malware by executing `java -jar static_analysis/static_analysis.jar samples/Pixstealer.apk $OUTPUT_PATH`.
3. Retrieve the malware analysis report at `$OUTPUT_PATH/Pixstealer.json`.

To avoid the possible security and privacy risks, we verified that the C&C servers the sample communicates with have been mitigated. The malware APK already contains C&C-loaded payloads that DVa's dynamic hooking framework extracted when the servers were live.

Running DVa against *Pixstealer* covers all three major malware analysis tasks claimed in the paper, namely detecting a11y malware's targeting victims, abuse vectors, and persistence mechanisms. The output should reveal:

1. Victim targets of the malware. This is organized as the package names of the victim apps or the system service that the malware targets. The victim target is shown under the "victim" key in the output JSON file together with its corresponding abuse vectors or persistence mechanisms. For the Pixstealer malware, you will see it targets the system "settings" app with multiple persistence mechanisms, the Nubank (com.nu.production), and the Inter&Co Financial APP (br.com.intermedium) apps with multiple abuse vectors.
2. Abuse vectors of the malware. This is shown under the "Abuse Vectors" key of the report and labeled as one of the a11y abuse vectors the malware uses to abuse each victim. Each abuse vector is accompanied by the concrete data-flow that leads to the execution of the vector. The vectors are all initiated from accessibility handlers of the malware and end with concrete a11y APIs. For the Pixstealer malware, you will see that it "steals credentials", and conducts "automated transactions".
3. Persistence mechanisms of the malware. This is shown under the "Persistence Mechanisms" key of the report and contains one of the a11y persistence mechanisms the malware adopts to hinder the user's removal of the malware. Each mechanism is accompanied by traces of the mechanism triggers. For the Pixstealer malware, it prevents the a11y permission revocation and prevents info look-up or uninstalling the malware.

A.4 Evaluation workflow

This subsection illustrates the major malware analysis results claimed in the paper. However, we are unable to release the malware dataset utilized in our research at this time due to ethical considerations. Consequently, users are required to obtain their own malware dataset for analysis.

A.4.1 Major Claims

(C1): DVa is able to identify 215 victim apps spanning seven categories abused by 4,291 Android a11y malware samples across 65 families. *Banking* apps are the most widely targeted category. This is proven by experiment 1 (E1) described in Section 5.1 of the paper and illustrated in Table 5.

(C2): DVa is able to extract seven categories of abuse vectors from a11y malware with an average of 13.9 vectors targeting each victim app. The most frequently adopted abuse vectors are *Steal Credentials* and *Auto Transaction* across most categories of victims. This is proven by experiment 2 (E2) described in Section 5.2 of the paper and illustrated in Table 5.

(C3): DVa is able to extract six categories of a11y-empowered persistence mechanisms from a11y malware. Most malware abuses a11y to prevent users from revoking a11y permission, looking up malware info / uninstalling malware, and disabling Google Play Protect. This is proven by experiment 3 (E3) described in Section 5.3 of the paper and illustrated in Table 6.

A.4.2 Experiments

(E1): [2 human-days + 14 compute-days + 200GB storage]: Execute DVa on a large scale to detect targeting victims of Android a11y malware.

Preparation: Collect Android a11y malware from online resources. To ensure an unbiased dataset, users should collect Android malware that requires accessibility permission randomly and across different malware families. Query the sample hashes against malware intelligence services to confirm their maliciousness and obtain malware family labels. Install the malware and run DVa's dynamic hooking framework on real Google Pixel 3 phones to aid the unpacking and loading of dynamic abuse payloads.

Execution: Run DVa's static symbolic execution component to extract targeting victims of the malware resolved by solving constraints to the execution of abuse vectors.

Results: DVa should output the package name of the victim Android applications the malware targets.

(E2): [2 human-days + 14 compute-days + 200GB storage]: Execute DVa on a large scale to detect abuse vectors of Android a11y malware.

Preparation: Collect Android a11y malware as described in E1.
Execution: Run DVa's static symbolic execution component to extract abuse vectors targeting victim applications. The abuse vectors are resolved by valid API sequence data-flows from the Android accessibility handler to a11y actions.
Results: DVa should output abuse vectors with which the malware targets each victim and a detailed data-flow path that constitutes the vector.

(E3): [1 human-days + 10 compute-days + 100GB storage]: Execute DVa on a large scale to detect persistence mechanisms of Android a11y malware.
Preparation: Collect Android a11y malware as described in E1.
Execution: Run DVa's static symbolic execution component as well as the dynamic hooking framework to extract a11y-empowered persistence mechanisms. The persistence mechanisms are resolved as valid API data-flow from the Android accessibility handler to a11y global action APIs with persistence mechanism triggers solved as constraints. They are also resolved as captured hooked a11y global action API calls after initiating the persistence mechanism triggers.
Results: DVa should output the a11y-empowered persistence mechanisms the malware uses.

A.5 Notes on Reusability

DVa has multiple scripts and components that can be utilized or extended to other Android malware analysis tasks. We list some examples below.

- **`A11yConstraints.java`**: A customizable script to detect a11y abuse vectors and resolving constraints. Can be extended to detect new abuse vectors when they are discovered.
- **`dynamicManager.py`**: A generic script that manages the execution of malware across multiple physical Android devices.
- **`singleDeviceManager.py`**: A generic and customizable script that utilizes *adb* to conduct multiple dynamic malware management tasks such as clearing device state, installing malware, querying a11y service info, initializing malware, granting permissions, uninstalling malware, etc. Users can easily insert or delete procedures that best suit their malware analysis tasks.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.