Building a Commit-level Dataset of Real-world Vulnerabilities
Alexis Challande, Robin David, Guénaël Renault

To cite this version:
Alexis Challande, Robin David, Guénaël Renault. Building a Commit-level Dataset of Real-world Vulnerabilities. CODASPY 2022 - 12th ACM Conference on Data and Application Security and Privacy, Apr 2022, Baltimore MD USA, United States. pp.101-106, 10.1145/1122445.1122456. hal-03477866

HAL Id: hal-03477866
https://hal.archives-ouvertes.fr/hal-03477866
Submitted on 21 Apr 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ABSTRACT
While Common Vulnerabilities and Exposures (CVE) have become a de facto standard for publishing advisories on vulnerabilities, the state of current CVE databases is lackluster. Yet, CVE advisories are insufficient to bridge the gap with the vulnerability artifacts in the impacted program. Therefore, the community is lacking a public real-world vulnerabilities dataset providing such association.

In this paper, we present a method restoring this missing link by analyzing the vulnerabilities from the Android Open Source Project (AOSP), an aggregate of more than 1,800 projects. It is the perfect target for building a representative dataset of vulnerabilities, as it covers the full spectrum that may be encountered in a modern system where a variety of low-level and higher-level components interact. More specifically, our main contribution is a dataset of more than 1,900 vulnerabilities, associating generic metadata (e.g., vulnerability type, impact level) with their respective patches at the commit granularity (e.g., fix commit-id, affected files, source code language).

Finally, we also augment this dataset by providing precompiled binaries for a subset of the vulnerabilities. These binaries open various data usage, both for binary only analysis and at the interface between vulnerabilities by aggregating those of a whole system. Our dataset, with a commit-level precision, includes vulnerabilities and their fixes in a unique dataset.

CCS CONCEPTS
- Security and privacy → Software security engineering: Vulnerability management.

KEYWORDS
security vulnerabilities; dataset; vulnerability research; patch detection; binary matching

ACM Reference Format:

1 INTRODUCTION
The CVE standard is used to publish vulnerability advisories through a common and universal mechanism. CVE databases [12, 23] present an overview of all the vulnerabilities in any system. To encompass such variety, they offer a set of features (CVE-ID, Common Vulnerability Scoring System (CVSS), description) and a method to describe impacted systems with Common Platform Enumeration (CPE) but no standardized way to report the corresponding fix. Data-driven vulnerability research relies on the availability of such information [8]. We aim to fill this gap by bringing together the vulnerabilities and their fixes in a unique dataset.

Android – one of the most deployed operating system in the world – makes it the perfect target to create such a dataset. It runs on heterogeneous devices like smartphones, car ECUs, smart devices or household appliances. Although some parts of the operating system are proprietary and closed, most of the system is open sourced through the AOSP. It is an aggregate of more than 1800 projects ranging from widely known projects to specific developments for a particular device. The projects cover the full spectrum of the functionalities that may be encountered in a modern system where a variety of components interact together (system tools, media framework, drivers and more).

Android is subject to security issues, and Google publishes monthly security bulletins listing CVE fixed in the last update. While the bulletins primarily target end users, they are also an information goldmine from a security perspective. Each bulletin contains a list of fixes affecting the operating system coming from three sources, AOSP, the Linux kernel and system-on-chip manufacturers. For every vulnerability entry, the CVE-ID, type, severity and the impacted AOSP versions are also listed. More importantly, it provides a direct link to the fixing commits for AOSP’s projects.

We leveraged this data to create the most exhaustive dataset of real-world vulnerabilities targeting a system as a whole. Previous datasets contain either synthetic code [7], hand-crafted vulnerabilities [9], or unrelated vulnerabilities in various software [16, 19]. Our dataset, with a commit-level precision, includes vulnerabilities targeting different languages (C, C++, Java). It brings coherence between vulnerabilities by aggregating those of a whole system.

We pushed our dataset a step further by providing pre-compiled binaries in both vulnerable and fixed state for a subset of these vulnerabilities (~600 vulnerabilities). This enables binary-based Source Code Analysis Tools (SAST), Dynamic Application Security Testing Tools (DAST) or CVE checkers to assess their capabilities at binary level.

Overall, our main contributions are:
• A dataset of more than 1,900 real-world vulnerabilities (based on CVEs) providing for each of them valuable metadata:
  – the vulnerability type (remote code execution, elevation of privilege);
  – the impacted programming language;
  – the patched and vulnerable versions at the exact commit-level.
• We augment this dataset by pre-compiling a subset of those vulnerabilities (600) to offer compiled binaries for multiple architectures (ARM, x86, X86_64, ARMv64).
• We make our data publicly available to the community to enable other researchers to test their research on broader data.

2 RELATED WORK

Previous vulnerability-related test suites and datasets have already been introduced in the literature. This section discuss previous datasets, their limitations and how our dataset addresses them.

The National Institute of Standards and Technology (NIST) Software Assurance Metrics And Tool Evaluation (SAMATE) project is dedicated to improve software assurance. The Juliet Test Suite [6, 7] is a collection of 64,000 synthetic test cases for the C/C++ language, testing 118 different Common Weakness Enumerations (CWEs). As part of standards, this test suite is extensively used by security vendors but suffers from its synthetic nature as generated programs are small and relatively simple.

In the LAVA [10] test suite, tailored for testing fuzzers, bugs are injected in different execution paths of binaries utilities from coreutils. LAVA only injects out-of-bounds memory accesses, not reflecting the complexity of real-world software bugs. The Cyber Grand Challenge (CGC) [9] sample set provides more bugs types, but the sample programs are relatively small and too simple to efficiently cover the wide range of existing binaries.

Instead of considering synthetic test-suites, the National Vulnerability Database (NVD) lists security issues affecting real-world software. Akram and Ping [3] presented a dataset of vulnerable functions and files gathered using NVD links to fix commits. Besides being discontinued, their dataset only lists vulnerable files and functions. This makes it hard to recover the context around those modifications and the fixed version is unavailable.

Li et al. [19, 20] created their dataset using vulnerabilities from some open-source projects. They used heuristics to detect the fix commit for a vulnerability. Thus, this information is computed while our dataset obtains it from its source, making it more reliable.

Vulnerable clone detection tools such as VFDETECT [22], VC-CFinder [25] or ReDeBug [16] also use a vulnerability dataset. Their dataset only contains some vulnerabilities for their tests and is unpublished, preventing its reusability.

Nappa et al. [24] studied the impact of shared code on vulnerability patching, but as they relied on CPE information, they were unable to be more precise than a version granularity.

Finally, other approaches use information from publicly disclosed vulnerabilities like the CVE benchmark from the Open Security Software Foundation [4], the work from Ponta et al. [26] or CVEFixes [5]. All these works are complementary with ours. For example, the CVE-benchmark targets vulnerabilities languages unsupported by our dataset (JavaScript/TypeScript), and only 3% of vulnerabilities found in CVEFixes are overlapping with our dataset.

3 ANDROID CVE DATA AGGREGATION

3.1 Bulletin Format

Since 2015, Google publishes monthly security bulletins [14] for Android. A bulletin is usually divided into two Security Patch Levels (SPLs), themselves divided into categories to facilitate the understanding of the bulletin: examples of categories are listed in Table 1. Finally, each category contains the list of vulnerabilities, with their CVE identifier, and the fixing AOSP updates. Additional information (i.e. affected versions, vulnerability severity) are also present. Table 2 shows an extract of the August 2021 bulletin.

Vulnerabilities listed in the bulletins are coming from various sources: AOSP itself, the Linux kernel, which is embedded inside AOSP, and system on chip manufacturers. For vulnerabilities affecting AOSP components, the bulletin contains a direct link to the fix commit. Publishing bulletins is not a Google specificity and other manufacturers like Samsung or Qualcomm also do it.

3.2 Architecture

We crawl Android Security Bulletins using a homemade tool named Roy. Its workflow is depicted in Figure 1. Roy starts by looking at the new bulletins since its last run. It is an iterative process and we never attempt to parse again previously analyzed bulletins. For each bulletin, Roy recovers the list of new vulnerabilities. When a link towards the fix commit exists, Roy also parses the changes provided by the commit (e.g. commit message, changed files...).
We developed an automated AOSP builder. It compiles the binaries where only binaries are available. To develop and test binary-level SAST, DAST or CVE-checkers, having a dataset containing binaries could prove itself useful.

AOSP is also a perfect target to provide pre-compiled binaries for the vulnerabilities: it is both open-source and presents a documented build system, while having commit-precise data.

4 AUTOMATED AOSP BUILDING

4.1 Motivation

Vulnerabilities affect both open-source and closed-source projects where only binaries are available. To develop and test binary-level SAST, DAST or CVE-checkers, having a dataset also containing binaries could prove itself useful.

AOSP is also a perfect target to provide pre-compiled binaries for the vulnerabilities: it is both open-source and presents a documented build system, while having commit-precise data.

4.2 Architecture

We developed an automated AOSP builder. It compiles the binaries at the commits just before and just after the vulnerability fix. Having binaries differing by a single commit, the one responsible for the fix, allows to pinpoint the changes induced by the fix at the binary level. The builder is depicted in Figure 2. As input, it uses a fixing commit id from Roy, builds the project with the fixing commit and without it. Finally, we only keep the binaries that differ between the two builds.

4.3 Building AOSP at a Specific Commit

Building AOSP at a specific commit is challenging for the following reasons. First, building projects in the past is difficult, as we have to reconstruct the whole chain of dependencies needed by the project (compiler version, libraries used...) at that time. AOSP self-containment, for both its toolchain and all its projects dependencies, makes the environment setup less difficult. However, finding precisely each dependencies version remains challenging. A second problem arises from either bogus commits preventing compilation of the project or fixes applied to former development branches. In this case, we cannot obtain binaries differing by precisely one commit and we use a different strategy (see Section 4.5). Finally, AOSP is a huge project of more than 80 GiB of source files. Building a single version of AOSP generates another 80 GiB of binary files in approximately 3 hours on a 56-core machines.

Our process to build vulnerabilities starts by looking at the latest branch including the fixing commit and setting up all AOSP’s projects to be at this version. This allows us to have a building environment working for AOSP in a whole and all the dependencies of the project. Then, we use two strategies to maximize the number of successful builds described in Section 4.5.

4.4 Build Diffing

A project may have multiple output targets, and AOSP’s projects depend on numerous projects themselves. Thus, building a single project may produce several binaries in the output directory. However, we are only interested in targets changed between the vulnerable and the fixed version. Since compilation is resource intensive, build systems implement an optimization, compiling for a target only if one of its build-dependencies has changed since the last run. We leverage this property for our build diffing system.

4.5 Vulnerability Building Strategies

We use two strategies to build the binaries of a project in both the vulnerable and fixed versions.

The first one is to checkout the project at a vulnerable commit, e.g. a parent of the fix commit, build the project, then checkout to the fix commit itself and rebuild the project. This is the preferred strategy as it is the most precise. However, it is unsuited if the compilation of the project in its vulnerable state fails.

The second strategy tries the opposite approach. It builds the project at the version state (one from Android) then reverts the fixing commit on the project before compiling again in a vulnerable state. If reverting the commit succeeded, this strategy works well.

4.6 Dataset Artifacts

For each vulnerability, we consider the build process to be complete if it produces the relevant binaries in both forms (fixed and vulnerable) for each of the four architectures (x86, x86_64, arm, arm64). We also keep the binaries with debug symbols (i.e. unstripped) if they are available.

Finally, we use heuristics relying on ctags universal [2] to guess the names of every function affected by a patch. These heuristics are unreliable for edge cases. Because parsing source code is done without running the preprocessor, functions affected by changes in macro are undetected. More importantly, compiler optimizations (e.g. function inlining, tail-call) may change the function layout and merge affected functions inside others.

Figure 3 presents an extract of the artifact for a precompiled vulnerability. We prefix every file by its SHA256 hash to prevent name collisions.
5 DATASET OVERVIEW

5.1 Commit-level Dataset

Our dataset contains 3,684 CVEs detailed in Figure 4. The closed source vulnerabilities affect Qualcomm, NVIDIA or Google components. We retrieved the fix commit id for 1,275 open source vulnerabilities (64%). We support gitiles [15], the platform used by Google for the versioning of AOSP, while other platforms such as GitHub, kernel.org or CodeAurora would need additional engineering effort. 35% of the vulnerabilities present in our dataset have a CVSS score of at least 9.0. Their CVSS score cumulative distribution is plotted on Figure 5.

For vulnerabilities with a fix-commit id, 70% of them affect a C-code source and 27% target Java code. The remaining 39 are fixes impacting other types of files (e.g. XML).

5.2 Precompiled Dataset

Among the 895 vulnerabilities targeting C/C++ code, we managed to propose precompiled binaries for 612 (68%) of them.

Based on file extensions, our dataset contains shared libraries (35%), object files (37%) and executables (13%). The exact repartition is depicted in Figure 8. The largest file is libv8 (1.6 GiB), the static library of v8, a JavaScript engine. Several metrics are also listed in Table 3.

<table>
<thead>
<tr>
<th>Category</th>
<th>Mean</th>
<th>Median</th>
<th>Standard dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstripped</td>
<td>12.3 MiB</td>
<td>2.7 MiB</td>
<td>40.2 MiB</td>
</tr>
<tr>
<td>Stripped</td>
<td>17.7 MiB</td>
<td>623.8 KiB</td>
<td>128.2 MiB</td>
</tr>
</tbody>
</table>
Table 4 lists the top 10 most common files found in the dataset. They mostly represent complex part of the operating system (e.g. wireless communications, media management) particularly targeted by attackers. For instance, the most common file is `libbluetooth.so`, found 953 times.

Table 4: Most Common Binaries

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>libbluetooth.so</td>
<td>953</td>
</tr>
<tr>
<td>bluetooth.default.so</td>
<td>748</td>
</tr>
<tr>
<td>libnfc-nci.so</td>
<td>650</td>
</tr>
<tr>
<td>libstagefright.so</td>
<td>421</td>
</tr>
<tr>
<td>net_test_btif</td>
<td>417</td>
</tr>
<tr>
<td>net_test_stack</td>
<td>299</td>
</tr>
<tr>
<td>hevcdec</td>
<td>268</td>
</tr>
<tr>
<td>libaudioflinger.so</td>
<td>242</td>
</tr>
<tr>
<td>libbinder.so</td>
<td>234</td>
</tr>
<tr>
<td>libmedia.so</td>
<td>229</td>
</tr>
</tbody>
</table>

Figure 9 lists the time delta between the vulnerability publishing date and the fixing commit. Notably, only 24 vulnerabilities were fixed after the creation of the CVE entry, while 122 were fixed more than 6 months before the publication. The average vulnerability is fixed 86 days before its CVE entry.

6 DISTRIBUTION

Our dataset is publicly available on GitHub. To ease data manipulation, we provide helper functions to interact with the dataset. Moreover, the precompiled dataset is also available online (117 GiB) and links are listed on the GitHub repository.

7 DISCUSSION

7.1 Applications

We provide this dataset to support data driven software security research. Examples listed below for each direction refer to existing research in the field. Nonetheless, we believe our dataset could enhance their works by having more data to test or reason about, while reducing friction as it is already compiled.

Patch Characterization. Our extensive same system vulnerability set enables researchers to extract data aiming at characterizing what a patch is at both the source and binary level. Our data can help draw an identity card of a security patch or bootstrap on patches detection.

Silent Fix Detection. A silent fix is a commit fixing a security hole without publicity (e.g. a CVE entry, a changelog mention). Silent patches create security risks for projects downstream as they are not aware of the importance of the new version. Being able to detect silent fixes could help maintainers to keep projects up to date, and reduce the risks for end users. Our dataset enables to train solutions both using the commit message (e.g. for Natural Language Processing algorithms) or analyzing the commits impact on the code.

Cross-architecture Binary Diffing/Matching. Our dataset provides the same binary for multiple architecture enabling further research on cross-architecture binary matching, where an application maps two code snippets from different architectures together if they have the same semantics. Moreover, some binaries in our dataset are present multiple times for different versions. They are interesting examples for cross-architecture binary diffing, where the objective is to find a mapping between functions in the first binary to the second one.

Patch Detection. Our dataset makes a perfect base to test tools aiming at solving the patch presence problem. By isolating the difference between the vulnerable and fixed version of the same binary, it enables the creation of signatures to detect if a target binary is patched.

7.2 Limitations

Android CVE Data Aggregation. An inherent limitation is the need for open source projects to retrieve the commit-id. Although open-source is prevalent in Android, many low-level components remain closed source. Our approach depends on the availability of security bulletins published by Google and their commitment in providing such information.

Data Quality. Our work implies that the commit referenced as a vulnerability fixing commit is complete, i.e. completely fixes the
vulnerability, and minimal, i.e. it does not fix any other problem nor add functionalities. Detecting if both assertions hold is out of scope for this paper and thus left unexplored.

Automated Compilation. Our automated compilation suffers from various problems and managed to compile only 612 vulnerabilities. Most issues stem from the synchronization problem between a project and its dependencies. However, each Android build has a manifest file listing the exact commit of each project during the compilation. Leveraging this information could help to solve the synchronization issues.

8 CONCLUSION

This paper shows how to build a dataset of real world representative vulnerabilities based on CVE identifiers leveraging Android Security Bulletins. This dataset is beneficial to the community by providing a common baseline vulnerability benchmark for further academic research and more robust vulnerability analysis or detection algorithms. We will continue collecting new data to keep our dataset synced with latest bulletins and CVEs.

The dataset is also augmented with more than 600 vulnerabilities precompiled for four architectures, enabling research on both source code, binary data or the interface between the two.

Our dataset is flexible so we can correlate it with other information sources or, at the opposite, tailor it for specific usages. Testing a SAST/DAST tool, a fuzzer, a sanitizer on a bug class (e.g. heap corruptions...), or conducting studies on vulnerability root causes or automatic repair are all possible usages of our dataset.

REFERENCES
[4] Bas van Schaik and Kevin Backhouse. FFs are cheap. show me the CVEs!
[17] Chen Liu, Qiang Wei, and Yan Cao. VFDETECT: A scalable code clone detection system based on vulnerability fingerprint. In 2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEIC), pages 548–553. IEEE.
[22] Zhen Liu, Qiang Wei, and Yan Cao. VFDETECT: A scalable code clone detection system based on vulnerability fingerprint. In 2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEIC), pages 548–553. IEEE.