On the Long-Term Reproducibility of Vulnerable Environments

Olivier Levillain[1][0000−0002−0558−5015], Nicolas Dejon[2][0000−0002−2802−0922], Clément Parssegny[1,3][0009−0004−2166−0881], and Sylvie Laniepce[2]

[1] Samovar, Télécom SudParis, Institut Polytechnique de Paris, France
olivier.levillain@telecom-sudparis.eu
clement.parssegny@telecom-sudparis.eu
[2] Orange Research
nicolas.dejon@orange.com
s.laniepce@orange.com
[3] Agence Nationale de la Sécurité des Systèmes d'Information
clement.parssegny@ssi.gouv.fr

Abstract. Software vulnerabilities fuel ever-growing cyberattacks, even years after their discovery and disclosure to the security community. Their study is thus essential to understand current threats and thwart future flaws; it includes the ability to reproduce attacks and more broadly to access vulnerable environments in the long run. This paper tackles the challenge of long-term reproduction of vulnerable environments.
We present a systematic methodology for creating containerized vulnerable environments using only publicly available data. We evaluate our methodology on a dataset of 142 CVEs, spanning four corpuses of CVEs and affecting the Debian distribution over a ten-year period, achieving a 41% reproduction rate. Our analysis reveals key factors that determine reproduction success or failure, and statistical insights about the evolution of the security posture. Moreover, we leverage our 58 successfully reproduced vulnerable environments as a ground truth to check the reliability of two standard vulnerability scanners, which detect 76% of the CVEs, thereby exposing blind spots for operational security teams. We release our methodology, our corpuses and associated container build descriptions as open-source artifacts to support long-term security research on software vulnerabilities.[4]

Keywords: Vulnerability Reproduction, Software Security, CVE.

1 Introduction

The ever-growing number of software vulnerabilities presents a critical challenge for companies; for example, the NVD (National Vulnerability Database), which

[4] The artifacts (the vulnerability metadata and their schema, and the container image description) are available as additional material on `https://github.com/Orange-OpenSource/vulnerability-open-knowledge`.

references CVEs (Common Vulnerability and Exposure) shows a steady growth (5k CVE in 2006, 10k in 2017, 20k in 2021, 40k in 2024 and 48k in 2025)[5]. Therefore, CVE management has become a significant operational concern facing the dilemma to allocate resources to remediate a massive number of CVEs, which are not oriented towards business development, while dealing with the risk of cyberattacks, which could severely impact business continuity.

Yet, the interest in producing and sharing reproducible environments to replay past vulnerabilities is scarce, especially in the long run, while this capability is essential to provide realistic test beds and real-world attack conditions in controlled environments. For example, it enables researchers to keep a history and a playground for their experimentation, and to study patches and the actual impact of vulnerabilities. It also helps to develop and verify countermeasures and security mechanisms. Furthermore, it allows training and education by the creation of realistic environments for CTFs or honeypots.

However, reproducing a vulnerability can be a challenging task. It involves general knowledge of diverse software ecosystems to gather relevant information about it, specific knowledge about the vulnerability itself to understand the expected behavior, and advanced skills in system administration to reproduce the vulnerable environment, including resolving dependencies for an old version. The access to validated and verifiable vulnerable environments also enables to assess the reliability of standard vulnerability scanners and better understand their limitations. In their work, Mu et al. [16] examined the *manual* reproduction of a corpus of 368 real-world vulnerabilities related to memory corruption issues, by security professionals from both academia and industry. While there is a connection with this work, notably in a common methodology, our work focuses on the step of the creation of the *vulnerable environment*, which means the test environment, and not on the *exploit production* per se. Indeed, unlike their approach, we do not engage in exploit development but instead leverage preexisting exploits solely to verify a successful reproduction.

In this paper, we aim to characterize the reality of long-term reproduction of vulnerable environments around three research questions:

- **RQ1**: To what extent is it possible to systematically reproduce a vulnerable environment with publicly available information?
- **RQ2**: What are the primary challenges that hinder the reproduction of vulnerable environments?
- **RQ3**: What critical gaps exist between public vulnerability intelligence and the operational realities of security teams?

To answer these RQs, we propose a systematic methodology to reproduce vulnerabilities. Our evaluation focuses on Debian environments with four relevant corpuses to our use case. The work has been conducted at least one year after the latest included CVEs to let time for exploits to become public after the disclosure of the 2023 CVEs. The dataset contains 142 CVEs affecting 64 different Debian packages, over the 2014-2023 period. We provide our database of vulnerable environments for the research community as an open source initiative.

[5] `https://nvd.nist.gov/vuln/search#/nvd/home?resultType=statistics`

Following the Background in Sec. 2, we present our systematic methodology in Sec. 3. Our methodology is empirically validated against CVE corpuses in Sec. 4, successfully creating 58 vulnerable containers out of 142. This dataset also serves as ground truth to assess the operational reality of security teams, such as the reliability of widely used vulnerability scanners. Finally, we discuss implications and limitations of our work, and outline promising future research directions, in Sec. 5. Sec. 6 presents related work and Sec. 7 concludes the paper.

2 Background

2.1 Vulnerable Environment and Exploit

A vulnerable environment is composed of (i) a vulnerable software piece at a given version, the target; (ii) a vulnerable environment, including all the target dependencies; and (iii) a configuration, to place the target in a vulnerable state.

An exploit is a proof of concept demonstrating a concrete exploitation of the vulnerability. It can be as simple as a command line invocation or require complex setups including external services. For example, CVE-2023-0795, is an out-of-bounds read in `libtiff`: we can trigger a crash by running the `tiffcrop` tool on a forged file with the proper options. On the other side of the spectrum, Log4Shell (CVE-2021-44228) requires building a malicious Java package, spawning an LDAP server to feed this package, and then trigger the vulnerability by including the malicious `ldap://` URL.

The reliability of an exploit may also depend on the target configuration, such as the activation of a given option in a service, or, at a lower level, a given version of a library when an exploit relies on the precise offset of a given symbol.

2.2 The NVD

For security researchers, vulnerabilities are often identified as CVEs (Common Vulnerabilities and Exposures) registered in the NVD (National Vulnerability Database). The NVD is a registry operated by the NIST (the National Institute of Standards and Technology), an agency of the US government. The NVD is synchronized with the running list of all CVEs assigned by CVE Numbering Authorities (CNA) of the CVE program and is thus the *de facto* reference vulnerability database.

Despite a rather structured data format, including links to the affected software components (the CPE, for Common Platform Enumeration, in the NVD terminology), it is sometimes difficult, from a CVE entry, to identify and install the precise version of the vulnerable software component. Moreover, CVE descriptions do not include any information on the relevant dependencies, let alone a formal identification thereof. Sometimes, CVE entries can include URLs to exploits or to a detailed description of the flaw.

2.3 DSAs and Other Linux Advisories

GNU/Linux Debian is a Linux distribution known for its stability, on which are based other popular distributions such as Ubuntu. The Debian Security Advisories (DSA) is a database maintained by the Debian project since 1997 to record the vulnerabilities affecting the Debian distribution. It contains classical advisories as well as metadata with many features, including the CVE entry number when applicable. Each year, around 200-300 DSAs are emitted, which in turn are related to around 1,000 CVEs.

The Debian security database also comprises, for a given vulnerability, the exact package names and information about the fixed versions. It is worth noting that, beyond the correct identification of vulnerable versions, Debian provides daily snapshots of their packages via `snapshot.debian.org`, which facilitates the reproduction of a vulnerable setup. However, DSAs do not include references to exploits, as can sometimes be the case with the NVD.

3 Methodology

This section describes our methodology to evaluate the reproduction of vulnerable environments. It presents the data collection strategy and the vulnerable environment reproduction process. We also describe here the selected corpuses of CVEs and the retained criteria for a verified reproduction. The evaluation of the corpuses has been dispatched among two security experts.

3.1 Target Ecosystem

In this paper, we restrict our work to software usually used in Linux containers, which gives us a realistic industrial perspective. This choice also corresponds to a more tractable abstraction level, since low-level software, which can depend on specific hardware or specific setups, are left out. It also allows us to exclude client-side applications, which usually come with configuration issues and may require user interactions. Focusing on Linux makes sense since it currently dominates the server market segment (63.73% of server market share in 2026).[6]

To make sure our setup is reproducible and well-defined for each vulnerability, we use Docker containers to describe the vulnerable environments in a concrete way (base image, software versions, etc.). When possible, we choose to consider binary packages from a specific Linux distribution, but we sometimes had to resort to manual environments using `git` repositories or published archives. It is worth noting that using code repository and specific commit numbers to identify a vulnerable version is less representative of the real-world deployment, but also less convenient and time-consuming.

We focus on Debian, because it is one of the most popular Linux distributions, especially in containerized environments. It also presents several advantages:

[6] `www.fortunebusinessinsights.com/server-operating-system-market-106601`

- many container images are based on Debian (among the 10 most popular images on Docker Hub at the time of writing, 7 have Debian-based tags);[7]
- Debian features reproducible builds (more than 90% of the packages are bit-for-bit reproducible), ensuring integrity assurance for our study;
- Debian has a strong process to manage security advisories, which is useful to link CVEs and exploits to actual software, with exact package names and versions which were affected by a given vulnerability;
- the Debian project hosts snapshots of every version of its packages and snapshots of repositories based on time. This last feature enables the retrieval of the versions of every package at a given time with a 6-hour precision.

As stated earlier, our use case is about containerized services, so we leave out security issues affecting client-side applications (e.g. browsers, games), and low-level packages (e.g. kernel, virtualization tools). We extracted all the Debian packages affected by security vulnerabilities (as referenced by DSAs and CVEs) over a ten-year period (2014-2023) and we manually labeled them according to our relevance criteria, resulting in a filtered set of 653 packages and a candidate pool of 4,674 CVEs. For our empirical evaluation, we constructed a representative dataset of 142 from this pool, organized into several corpuses. The methodology for the construction of these corpuses is presented in Sec. 3.3.

Table 3 (in appendix) shows the breakdown of the packages we left out, and the proportion of CVEs affecting each category, for the 2014-2023 period.

3.2 Vulnerable Environment Reproduction Process

To reproduce the CVEs, two security experts followed the following procedure. They were instructed to only use existing exploits, possibly with light modifications, but never develop their own. Once an expert had produced a description of a CVE, the second one reviewed the artifacts for cross-verification purposes.

Data collection Since we focus on Debian, we use the available metadata in the DSAs, allowing us to map a CVE to a source package and its version. To this aim, we interact with Debian security resources from the Debian Security Tracker to link the CVE to a specific DSA, which will help us pinpoint the corresponding package and the vulnerable versions for the release we target.

We then use package information from the Debian archive to find a timestamp when the first snapshot with the vulnerable package was taken.

Environment creation and setup This step requires to instantiate the information collected during the previous step into an executable vulnerable environment. It consists in: a) setting up the base environment for the container and b) installing and configuring the software at the vulnerable version.

[7] The authors of [23] also chose to focus on the Debian ecosystem for its popularity when the studied the prevalence of vulnerabilities in Linux container images.

To set up the base environment, we use Docker Hub which maintains a great variety of tags for Debian. Even if the granularity of these images is a lot coarser (monthly images) than the granularity of `snapshot.debian.org` (several snapshots per day), it is easy to find an image containing a Debian system which is as close as possible to the vulnerable environment from the past.

The next step is to install the vulnerable target software itself. There are two strategies to handle this *installation phase*: a) use existing binary packages, b) rebuild the vulnerable component from its source. In the first case, once the base image has been chosen, we simply rely on the snapshot infrastructure provided by the Debian project, which works as a time machine, to install the vulnerable target software, with its actual dependency graph at the chosen timestamp. To do so, we use the `apt` (the Debian package manager) with a `sources.list` file pointing to the timestamp corresponding to the chosen snapshot. When using binary packages fails, e.g. because the installed version is not actually vulnerable[8], we must follow the second strategy and resort to installing the vulnerable component from its source code (a Git repository or a release archive). Doing so requires finding the correct version (git tag or commit) and making sure the dependencies are properly set up, which might not be obvious.

Finally, the target software may require some form of *configuration phase* to conform to the studied CVE, ensuring the exploit can be successfully triggered and the effects of the vulnerability be observed. This phase involves reading the advisory and the related resources to build proper configuration files and invocation scripts when needed.

Exploit search and reproduction validation We know from Mu et al. [16] that exploit reproduction is a difficult and time-consuming task, so we only consider existing exploits on the clearnet, excluding dark and deep web resources, as well as home-made exploits. This approach correspond to a moderate-level attacker having access to the web.

Concretely, we looked for exploits using the Google and DuckDuckGo search engines, as well as sources such as Exploit-DB, the NVD, GitHub repositories including CVE identifiers, PacketStorm, OpenWall OSS Security mailing list.[9] We excluded any information that was not open, for example forums requiring the creation of accounts, even if this step was free. While sometimes a bug reproducer was found without direct link to a CVE, the description and the data of publication made it possible to link back both. Some CVEs were illustrated by several exploits, in such cases we selected the first working exploit that looked the simplest to set up with at best an already proposed Docker image with required exploitation resources and instructions.

If all steps succeed, the vulnerable environment reproduction is validated. Otherwise, the expert explains the reason of the failure that prevented the reproduction. This process resembles the 4-stage workflow methodology of Mu et

[8] It can happen if the binary package is compiled with the wrong options, or when the advisory wrongly reports Debian was affected by the corresponding CVE.

[9] The search keywords included the CVE ID, “exploit”, “PoC”, “vulnerability”, etc.

al. [16] for vulnerability reproduction (report gathering, environment setup, software preparation and reproduction), however it is different in the sense that the focus is **not** on the *vulnerability* itself but on the vulnerable *environment* which involves subtle differences at each step.

3.3 Corpus Selection

In this work, we focus on reproducible userland CVEs, leaving out kernel exploits, that can be reproduced in a Docker container context based on Debian. Out of the candidate vulnerabilities described in Sec. 3.1, we created 4 corpuses.

`10Y-50` contains 50 relevant CVEs affecting Debian during the 10-year period spanning 2014-2023 ($\approx$ 1% sampling rate for the same period). We chose to include several popular vulnerabilities, e.g. Heartbleed, ImageTragick, Baron Samedit and PwnKit, and we sampled vulnerabilities to balance various criteria (different severities, varied affected packages, broad spectrum of dates).

`Recent-50` contains 50 relevant CVEs affecting Debian during the 4-year period spanning 2020-2023 ($\approx$ 3,5% sampling rate for the same period). The selection was similar as the one used for `10Y-50`.

`Pop-2023` is the set of all CVEs affecting a relevant package in the top 1k most popular Debian packages as provided by the Debian project during the whole year 2023, totaling 74 CVEs.[10]

`Cage4Deno` is a small corpus provided by Abbadini et al. [1], which affect various programs (e.g. FFmpeg or ImageMagick) to test a kernel-level sandbox. The original corpus contains 15 CVEs, but only 10 map to our candidate pool of CVEs. This corpus enables to assess the reproducibility of research on vulnerable environments over time. It is worth noting that the companion Git repository does not include the actual recipes to reproduce the studied CVEs.

The two first corpus are designed to sample vulnerabilities selected by the authors to cover some variety in time, severities, or affected packages, whereas the third one was designed to be an objective corpus. Finally, the last corpus allowed us to confront our methodology to a vulnerability corpus chosen by another research team, to check whether we could adapt to such a selection.

3.4 Evaluation Criteria

Our systematic vulnerability study encompasses a manual study of each CVE. The analysts rate the reproduction according to a comprehensive data model to ensure repeatability. We use a YAML file structured into four sections:

- CVE enhancement: data extracted from the NVD and DSA public advisories, e.g. description, impact and publication dates, as well as additional information provided by the analyst, such as missing details, a rating score about the advisory information quality, and the patch timeline.

[10] We could not use the "popcon" data for older CVEs, since the data are not versioned, and packages change names over time, which could have led to inconsistencies.

- Vulnerable Environment Specification: details about the vulnerable container image, including the Debian release, the targeted package and its vulnerable version, the installation method (using `apt` or from sources) and environment complexity (e.g. the number of required files or services);
- Exploitation Specification: information about the exploit process to trigger the vulnerability, such as the source of the exploit, its reported date of public release, and exploit configuration complexity like files or services;
- Reproduction Validation: information about the status of the vulnerability reproduction, including the reason of reproduction failure if any.

The reproduction is considered **validated** only if the complete expected impact is shown by a **successful exploitation of the vulnerability**. This implies that when only one aspect of the expected impact is demonstrated (e.g. a crash when expected code execution), the reproduction is only partially validated, and when no exploit is found, the reproduction is not considered successful. Indeed, a vulnerable environment where we cannot provably run an exploit is in practice indistinguishable from a non-vulnerable environment.

4 Results

In this section, we support answers to the RQs with both quantitative results based on statistics and qualitative results based on a chosen set of packages. Table 1 is an overview of the quantitative results over the corpuses. It is worth noting that several CVEs are present in multiple corpuses, which explains why the total of 142 unique studied CVEs in the table is not the sum of the lines.

Table 1. Summary of the Reproduction and Detection Rates over the Studied Corpuses. The Reproduction Rate counts vulnerabilities that were fully reproduced (or at least partially reproduced, in parenthesis). The Detection Rate counts the images correctly flagged as vulnerable by scanners, among the fully reproduced CVEs.

Corpus	Relevant CVEs	Reproduction Success	Detected by Scanners
`10Y-50`	50	44% (50%)	73%
`Recent-50`	50	44% (52%)	68%
`Pop-2023`	74	34% (57%)	76%
`Cage4Deno`	10	80% (100%)	88%
Total	142	41% (57%)	76%

4.1 RQ1: To what extent is it possible to systematically reproduce a vulnerable environment with publicly available information?

The reproduction success rate of vulnerable environments yields differently depending on the selected corpus. Indeed, for the `10Y-50` and `Recent-50` corpuses,

almost half the CVEs were at least partially reproduced, among which 22 CVEs fully reproduced for both corpuses, i.e. 44%. The results for `Pop-2023` are more blurry, since we could at least partially reproduce 57% of the CVEs (which is consistent with the other results), but only 34% were fully reproduced.

It is worth comparing these figures to similar results publicly available. First, [10] states that around 4% of CVEs published between 2013 and 2019 had an associated public exploit within a 1-year period; this result seems at odds with our work, but it is worth mentioning we focused on a "relevant" subset of CVEs affecting Debian, whereas CVEs in general cover a broader spectrum (we consider less than 5k CVEs for a 10-year period, instead of 75k CVEs over a 7-year period). A 2025 report from Qualys[11] states that 26.5% of the 26k CVEs published in 2023 came with a proof of concept. Again, the figure may seem low, but the focus on a subset of the 2023 CVEs (a little more than 200 CVEs are linked to a Debian Security Advisory, to be compared with the 26k overall figure).

Finally, the `Cage4Deno` corpus contains 10 CVEs that were used in a research paper evaluating a security mechanism. Even if the authors did not provide artifacts to reproduce the vulnerable environments they used for their evaluation, their experiments relied on the actual reproduction of the CVEs. Our analysis shows that it is actually possible to reproduce them (at least partially).

During our study, we rated the information quality from the CVE description using the following values: Detailed (very detailed descriptions), Minimal (comprising the vulnerable versions and highlights the vulnerable function), Evasive (ambiguous and misleading information), Missing (lacking critical information) or Incorrect (erroneous information). While no CVEs of our dataset were tagged as Incorrect, we observe a correlation between reproduction success and the CVE information quality. Specifically, the likelihood of success gradually decreased as information quality degraded, from Detailed to Missing, as shown by Figure 1.

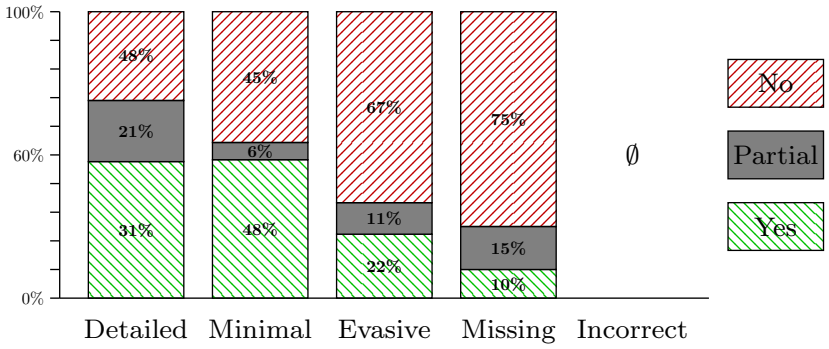

Fig. 1. Reproduction status per CVE Information Quality level.

[11] `https://blog.qualys.com/vulnerabilities-threat-research/2023/12/19/2023-threat-landscape-year-in-review-part-one`

The following CVE example illustrates how fragile concrete exploitation can be because a vulnerability depends on more than one cause. Another example is given in Appendix A.2, to show that a kernel-level change can suddenly thwart a whole class of attacks.

CVE-2020-7247: RCE in `opensmtpd`. The OpenSMTPD mail server is vulnerable to a shell injection, which can lead to a Local Privilege Escalation or even a Remote Code Execution as `root` if the server is publicly exposed.

Debian security metadata indicate that the vulnerability affects Debian 9 (the `opensmtpd` 6.0 branch) to 11 (the 6.6 branch), but the issue is actually only exploitable in Debian 11. Indeed, the input sanitization is also missing in the earlier release, but the dangerous path leading to an actual shell injection was only introduced in `opensmtpd` 6.6. Metadata can thus sometimes be misleading or incomplete, and that it is important to confirm that a bug can be exploited.

4.2 RQ2: What are the primary challenges that hinder the reproduction of vulnerable environments?

We now analyze the reasons of reproduction failures, summarized in Table 2. Conversely to our results in RQ1, across all corpuses, lack of reproducibility is observed for an average rate of 59%.

Table 2. Non Reproduction Reasons.

Reason for Failure	Not Reproduced	Partially Repr.
Exploit Not Working	9	0
Incomplete Exploit	0	6
Memory Sanitizer Only	0	15
No Exploit Available	46	1
Non Functional Container	2	0
Missing Information	4	1
Total	**61**	**23**

Of the 142 we studied, we completely failed to reproduce 61 of them (43%): we could not observe any aspect of the expected impact. The main cause of failure (75% of the CVEs) is the absence of a public exploit. Other factors are non-functional exploits (15%) and issues to find the vulnerable component or configuration (7%). For 23 out of 142 (16%), a vulnerability flaw was confirmed, but the full expected vulnerability impact could not, however the reasons are different than for complete lack of reproduction. The main cause of failure (65%) is that the PoC requires the vulnerable component to be instrumented with compiler options, such as ASAN (Address Sanitizer), to expose the vulnerability

(crash, memory overflows). We also found that for 26% of these partially reproduced CVEs, the exploit was incomplete, e.g. provoking a crash instead of code execution: we could only exhibit an unexpected, potentially dangerous behavior, instead of the full impact. We identify two categories of reasons which prevented us from validating environments according to our criteria.

Challenges in the construction of vulnerable environments. This first category of failures only amounts to a small number of cases in Table 2, either when we could not find an adequate base image (Non Functional Container), as public container registries do not provide such images after several years, or when we failed to properly configure the target (Missing Information).

Challenges in the exploitation phase The exploitation phase fails when we are unable to confirm the presence of the CVE using a PoC, despite the successful creation of a (hopefully) vulnerable environment. We identify three primary causes, as shown in Table 2: (i) the absence of a public PoC, which makes it impossible to empirically validate the vulnerability of an environment; (ii) non-functional or incomplete PoCs, which usually depend on specific environment details (e.g. memory layouts, library versions, OS distribution); and (iii) a dependency on unofficial and unreleased environments, such as AddressSanitizer or other compiler options, which alters how programs run to detect memory corruption errors. Let us illustrate this with partially reproduced CVEs.

CVE-2023-0795 to -0804: A series of DoS issues in `tiff`. When we worked on the `Pop-2023` corpus, we analyzed DSA-5361-1 about `tiff`. This advisory is associated to 10 different CVEs, which all correspond to memory corruption issues such as heap-based buffer overflows. In the GitLab tickets reporting the bugs, the steps to reproduce require to recompile the package with `-fsanitize=address`, to activate code instrumentation and detect memory errors at runtime.

For 7 out of 10 of this series of CVEs, we actually had to recompile the project from source to witness a crash, whereas the corresponding Debian package silently accepted the proofs of concept, hence the "Partial" verdict for these CVEs. However, for 3 of them, the PoC also led the vanilla Debian package to a Segmentation Fault, so we marked them as *fully* reproducible DoS.

Such situations are typical of bugs found by fuzzers in conjunction with memory sanitizers. The bugs are real, but proving the actual exploitation thereof (even for a DoS) depends on many factors and is not always easy to determine.

CVE-2022-0529 and -0530: Non-confirmed RCE in `unzip`. In 2022, two memory issues were reported against the `unzip` utility: a heap-based buffer overflow (CVE-2022-0529) and a NULL pointer dereference (CVE-2022-0530). Both issues can lead to a crash, but the CVE description in the NVD states that "[they allow] an attacker to input a specially crafted zip file, leading to a crash or code execution". We thus expect the impact of these CVEs to be more than a DoS.

A GitHub repository[12] contains a proof of concept for both CVEs. However, they only produce crashes, not code execution. The documentation in the repository is less affirmative than the NVD entry: "[They allow] an attacker to perform a denial of service and possibly [open] up other attack vectors."

CVE-2020-11501: Partial Reproduction without a PoC. Due to a trivial error in a check in a conditional branch, where the tested condition is the opposite of the intended one, GnuTLS does not contribute its random value to the DTLS handshake, filling the `client_random` field with zeroes. This mistake breaks DTLS security gurantees and paves the way to complex cryptographic attacks possible, e.g. replay attacks or plaintext recovery.

Exploiting this to recover plaintext is complex, and there was no public exploits. However, by merely capturing the traffic when a vulnerable GnuTLS client connects to a DTLS server, we can witness that the client random field is filled with zeroes. Thus, even if we cannot exploit CVE-2020-11501, a simple command line can prove the presence of the bug. This justifies the "Partial" reproduction status with the "No Exploit Available" reason.

4.3 RQ3: What critical gaps exist between public vulnerability intelligence and the operational realities of security teams?

We now take the perspective of an operational security team facing industrial challenges. The team relies on public data and has access to standard CVE detection tools to scan their container images. This research question explores their ability to prepare their systems against attacks, balancing between CVE disclosure, patching ability, and attackers' exploit weaponization opportunity.

Disclosure and Patching Timeline We first focus on the public CVE disclosure and the patching timeline, so when vulnerabilities become publicly known and patches become available. Our results show a wide disclosure embargo period between the day an official patch is released and the disclosure on NVD, reflecting the responsible disclosure process of software projects: 33% of CVEs are disclosed on NVD 30 days after the official patch (median embargo: 18 days). Still, reactivity is expected from security teams for certain types of vulnerabilities when disclosure is released rapidly in order to quickly patch systems: 19% are disclosed the same day as the official patch release, and 19% in the week after. Interestingly, we observe some correlation between CVSS score and disclosure lag, suggesting that higher-severity vulnerabilities tend to have shorter embargo periods (median embargo of 7 days Critical CVEs, but 19 days for Medium).

However, public disclosure does not always translate immediately to available patches: 7% of CVEs are disclosed before any official patch exists. Notably, pre-patch disclosures show higher CVSS scores (mean 8.0 vs 6.8 for post-patch).

From the Debian perspective, taking the official patch as the reference date, the temporal glitch differs drastically between projects. Overall, the fixed version

[12] `https://github.com/ByteHackr/unzip_poc/`

of a package is released within a week for 31% of CVEs, reaching 61% within 30 days. This means security teams should still expect that 39% of CVEs remain unpatched in the Debian package 30 days after the official patch release, leaving systems unprotected during this period. Nevertheless, Debian shows a significant speed to provide a patched version for Critical/High CVEs compared to Medium/Low severities, respectively with a median lag of 6 days for the first (Critical: 3, High: 12) and 34 for the latter (Medium: 34, Low: 10).

Exploit Availability and Weaponization While defenders wait for patches, attackers search for exploits. In this work, we have studied exploit availability, the sole condition to validate a vulnerable environment. A critical finding is that 42% of proof-of-concept (PoC) code was available before or simultaneously with NVD publication. We can expect that most of these PoCs have not been weaponized into exploits and only released as part of a responsible disclosure, however, we could also imagine attackers gaining the same knowledge during the same period.

For the rest, 20% of our verified exploits (16/81) appear more than 7 days after NVD disclosure, which are likely weaponized exploits. For these considered weaponized exploits, 19% of them appear within 30 days of disclosure, and security teams should expect a median weaponization period of 196 days after CVE publication. This provides defenders with a critical window for patching before weaponization really occurs. This also means that, at the date of realization of this work, we are located after that critical exploit release period, thereby consolidating our results on the availability of exploits. However, counter to our intuition, we find no significant correlation between CVSS severity and exploit availability, suggesting that CVSS alone is insufficient for CVE prioritization.

Package popularity shows minimal impact on exploit availability: the top 10 most CVE impacted packages have a 41% exploit rate, same as less CVE impacted packages. However, the package's category matters significantly: Compression/VectorGraphics/Directory/VideoCodecs/Document packages (such as `zzlipb`, `librsvg`, `openldap`, `libde265`, `ghostscript`) have a 75-100% exploit rate, while DNS/NetworkServices have 12% (e.g. `bind9`, `ntp`) and Database (e.g. `postgresql`) packages even have 0%. This can be explained by the fact that network services require more work to actually set them up and configure them properly.

Debian Vulnerability Windows At the time of NVD disclosure, Debian had already provided a fixed version for the vulnerable package 39% of the time. If we only consider functional exploits released after the NVD disclosure and after the release of an official patch (post-disclosure exploits, post-patch period), our results show that 71% of Debian affected packages were protected against them at the time they were publicly released (other exploits could be existing before the exploit we kept for the study, so this is an upper bound). This means at least 29% of Debian affected packages were vulnerable when a post-disclosure exploit was released. Furthermore, functional exploits are sometimes in the wild

without a fix: still 14% of Debian packages remain vulnerable 7 days after the NVD disclosure, going down to 0% after 30 days. This represents the attacker's time window during which the exploit can be weaponized successfully.

Exploit Source Functional exploits are mostly found on GitHub (45%), then on specific blogs (10%), GitLab (12%) which are usually the package's project Git), security advisories (9%), bug trackers (7%), and the rest on other exploit databases (only 7% of exploits are taken from Exploit-DB), or academic papers.

Container Vulnerability Scanner Reliability Lastly, we study container vulnerability scanners used by operational security teams to detect CVEs. We have run two standard industrial-grade tools, Trivy 0.69.3 (with a database retrieved on March 4th 2026) and Grype 0.109.0, over all our successfully reproduced vulnerable images. For each vulnerability, we run the tools on the image, and we check that (at least) the expected CVE is found by the tool.

Even when aggregating the findings from both tools, 24% of the vulnerable versions were NOT detected by either of them, despite our manual verification of a successful exploitation on the same container images, thus achieving only a 76% detection rate at maximum. There is a small bias for CVEs where we had to manually install the vulnerable component from source code (Git repository or archive), for which the vulnerability scanners are usually unable to find proper CVE artifacts.[13] The effect is negligible on our conclusions since we only have reproduced 4 CVEs using the source code, the rest with `apt`.

Surprisingly, the tools do not agree on one successfully reproduced CVE: Grype marks CVE-2014-8129 (an out-of-bounds write in the `tiff` package) as present whereas Trivy misses it.

5 Discussion, Limitations and Future Work

5.1 Reproduction success factors and challenges

Our study reveals critical insights into the reproduction of vulnerable container environments and the quality of publicly available information. From our observations, we draw the following conclusions. **Success factors** are 1) the vulnerable package version is clearly identified, 2) a functional exploit has been publicly released and 3) the CVE is described in details, which facilitates the configuration of the target environment. **Causes of failures** are 1) an exploit is missing, non-functional, or incomplete, 2) too many missing information from different sources: vulnerable version, vulnerability description, configuration of target, configuration of exploits, and 3) the impact might not be totally observed if the outcome is produced by memory sanitizers or fuzzers.

[13] Interestingly, Grype successfully identify 4 images as vulnerable, even if they were built from source. These 4 CVEs could not however be reproduced.

5.2 Reproduction Efforts

Since there is a natural direction towards large-scale experiments, it is worth discussing the manual effort required for each vulnerability reproduced. For this work, we used our open-source tool named DECRET[14], which was specially crafted for this study to automate the build of the vulnerable environment using Debian security metadata and existing PoCs from Exploit-DB. Even if the produced artifacts usually required adjustments, DECRET gave us a first skeleton. Then, both the adjustments and the review were done perusing various open sources. We did not measure exactly the time for each CVE, but the reproduction usually took between a few minutes up to 5 hours in rare cases (which is the limit we gave ourselves for a given CVE). Review was usually straightforward and took 10 to 15 minutes.

5.3 Scope and Generalization

By construction, our study does not represent the entire CVE population, but only a fragment specifically affecting Debian distributions through vulnerable packages relevant to container environments. In particular, even if the `Pop-2023` corpus corresponds to an objective selection for the most popular Debian packages, it only corresponds to 5 % of the CVEs that affected Debian in 2023. It is thus debatable whether we can generalize our findings to the whole Debian ecosystem, let alone other OSes. However, it is worth noting our potential bias is focused on an operational approach (containerized services, popular packages) which should correspond to what most security response teams are facing.

5.4 Dependency on the Public Information Ecosystem

Our methodology, tightly coupled with public data (DSAs, Debian snapshots, PoCs), leaves any commercial, secret and manual exploits out of scope. This work is then highly sensitive to the public data quality from which we derive two major weaknesses in the ecosystem: information gaps and information volatility.

Information gaps Our work studies the quality of public data, when *data quality*, like incorrect or missing information. For example, CVE descriptions may sometimes be deliberately kept to disclose as little information as possible (or even no information at all). An extreme example of such an entry is CVE-2015-4819, which is described as an "unspecified vulnerability in Oracle MySQL Server 5.5.44 and earlier, and 5.6.25 and earlier, [which] allows local users to affect confidentiality, integrity, and availability via unknown vectors related to Client programs.", with no relevant links in the NVD database.

We also rely on *data availability*, such as public mainstream search engines, to seek information and PoCs. In this process, we could miss the attack habits of more advanced attackers, knowing specific websites where exploits could be

[14] `https://github.com/Orange-OpenSource/decret`

found that might not be indexed by these search engines. The absence of a verified exploit in our database should then not be perceived as a low-priority issue for defenders, as attackers might possess private exploits. Furthermore, conducting a full sequence of attacks also raises challenges limiting the interpretation of our results from an attacker's perspective.

Information volatility Our results date back to mid-2025 on CVEs affecting the year 2023 and prior. However, the public data on which our work is laid down is evolving, which means data could be augmented, or, on the contrary, be removed from the public sources that we consider, for example when a snapshot infrastructure closed because of products' (planned) end-of-life. This remark is also to be considered when exploits are hosted on user-controlled websites (blogs, personal repositories), that could disappear at any time. Conversely, exploits could be produced any time in the future, since we showed in Sec. 4.3 that at least half of the likely weaponized functional exploits (exploits that appeared at least 7 days after the NVD disclosure) should have been published at time of writing based on the 10-year trend. It is thus hard to state an actual reproduction status which should be maintained in time for accurate and up-to-date results.

5.5 Ethical Considerations and Responsible Disclosure

Our methodology is designed to adhere to established ethical standards. We rely exclusively on publicly available information and as such, do not engage in vulnerability discovery or exploit development. Neither did we buy information or exploits for this article.

The PoCs, central to our methodology for validating the vulnerabilities, could also be used for malicious purposes. Because of the risk of misuse, we choose not to redistribute them in our open source artifacts. Instead, to ensure transparency and verification, we provide metadata and links to the original, publicly accessible sources. The security community can then have confidence in our data and strengthens their defenses, without our work directly facilitating attacks.

5.6 Future Work

Expanding the scope of the corpus. Our vulnerability corpus should be extended to include more CVEs and to other Linux distributions or other open-source OSes (e.g. OpenBSD) as soon as the relevant metadata and snapshots are available.

Automation and AI-assisted reproduction. Our open-source tool, DECRET, has areas for refinement and future work will focus on improving its reliability and thus increase the automated successful environment creation rate. Recent advanced AI technologies could help to improve DECRET and could also be leveraged to produce exploits, or rewrite existing ones, that are "safe": exploits designed to prove a vulnerability while ensuring a safe execution (terminate gracefully) or containment to eliminate side effects. AI could also help to supplement every manual effort that was provided in the study and palliate the difficulties of

reproduction by curating metadata and homogenizing the cross-verification, fixing container errors, assisting analysts in understanding the vulnerable package and how it works if unknown or explaining the causes of the vulnerability.

Recent work in this direction has been published: CVE-Genie [22] has demonstrated the potential of end-to-end agentic workflows to automate the generation of vulnerable environments and produce exploits. They reproduced 428 CVEs out of the 821 recent vulnerabilities of their 2024-2025 corpus, at a relatively low average budget of $2.77 per CVE. It would be interesting to test this methodology on a wider corpus including older CVEs like ours.

Long-term reproducibility. We identified temporal volatility as a challenge because public artifacts disappear over a ten-year period.

A research direction is to periodically re-evaluate the same CVE corpus which would allow us to quantify the rate of information loss. Future work could ensure a durable vulnerability reproduction archive, which could start with our released corpuses, on the model of existing projects about software preservation such as Software Heritage [7]. To this aim, an interesting extension of our work would be to build a proper archive to keep the different elements required by our methodology (base container image, package repositories, but also relevant kernels and external source repositories).

Moreover, designing a binary-package-based distribution infrastructure offering bit-to-bit reproducible builds, which would guarantee immutable snapshots, is still challenging. Promising recent findings from Benedetti et al. [4] confirm reproducibility issues in modern package ecosystems could be easily fixed.

6 Related Work

Recently, Li et al. [14] conducted a systematic mapping study on the use of vulnerability databases in research which highlights the scarcity of work on vulnerability reproduction. The authors identified only one paper which focuses on vulnerability reproduction as in our work. In this paper, Huang et al. [11] propose to leverage information on real vulnerabilities to automatically build DVE (Deliberated Vulnerable Environment) containerized images verified as vulnerable to a given vulnerability. However, their evaluation is limited to 4 CVEs and their approach lacks a systematic data collection pipeline. In contrast, we provide a systematic, scalable methodology for long-term vulnerability reproduction and validate it on a diverse corpus of 142 CVEs.

Anwar et al., Dong et al. [3, 8] investigate the challenge of data quality and consistency through large-scale studies in the NVD, which revealed significant inconsistencies across key metadata, including misleading publication dates, ambiguous vendor and product names, incomplete CVSS, and unstructured text from CVE summaries. While they focus on detecting and quantifying the inconsistencies, we address the subsequent practical challenge of using this imperfect metadata to successfully construct and validate vulnerable environments.

Another line of research analyzes source code repositories to understand the vulnerability life cycle, like identifying Vulnerability-Contributing Commits

(VCCs) [15, 17], locating patch commits [6, 18], and curating large datasets like Vul4J [5] and Big-Vul [9] that link CVEs to specific commits. However, these source-code-centric approaches do not address our challenge of reproducing vulnerabilities from pre-compiled packages. Differently, several studies propose datasets with a limited number of CVEs (15-43) to evaluate security tools [21, 12, 1]. However, these works do not build vulnerable environments, they rarely provide the actual reproducers of the various vulnerabilities, nor do they validate their data by triggering the vulnerabilities, as we propose in this work.

Mu et al. [16] quantified the high cost of manual reproduction of memory corruption vulnerabilities. Security experts spent an average of 5 hours to reproduce a single CVE, using only public information and their expert knowledge. They achieved a 62.5% reproduction rate on the studied 291 CVEs. This work is complementary to ours: they focused on expert-driven task of exploit creation, we focus on the assisted creation of the vulnerable environments and target a broader spectrum of vulnerability types.

Akasaka et al. [2] describe a project similar to ours with `vultest`, which generates a vulnerable VM from a YAML file. However, their approach lacks expressiveness in their description language, their artifacts cover a small corpus (6 CVEs) and rely on a base VM image, which are not available anymore, making their work impossible to reproduce in practice. Our work covers a much broader set of CVEs, with a more expressive description language (`Dockerfile`s); we also addressed the volatility issue by relying on the Debian `snapshot` infrastructure.

Our focus on userspace applications is complementary to the recent work of Ruan et al., which proposed KernJC [19], a tool to automatically rebuild Linux kernels to reproduce vulnerable environments.

Using real-world CVEs also has applications in the training dimension. There are some academic papers studying how to create CTF challenges including real-world vulnerabilities [20, 13]. Merging these approaches with our database could be interesting to help automate the creation of unique challenges.

Beyond academia, community-driven platforms also provide vulnerable systems, such as VulHub and VulnLab.[15] for containers and VMs, designed for security training. However, they usually represent manually crafted environments to illustrate a specific attack, without giving transparent steps to reproduce the containers. In contrast, our work provides a systematic methodology to build and validate environments for a large and diverse corpus of CVEs.

7 Conclusion

In this paper, we investigated the reality of reproducing vulnerable environments, often overlooked in security research. We applied our methodology in a containerized context, on a dataset comprising 4 corpuses for a total of 142 unique userland CVEs affecting the Debian distribution over a decade, obtaining an average reproduction success rate of 41%. We identified information accuracy

[15] `https://vulhub.org/` and `https://www.vulnlab.com/`

and PoC availability as the primary factors governing success in the long run, and conversely, lack or incorrect information on the target configuration or PoC, and missing or non-functional PoCs as major causes of failures. Furthermore, our 58 successfully reproduced environments served as ground truth to evaluate standard vulnerability scanners, which failed to detect 24% of these confirmed vulnerabilities, exposing significant blind spots for operational security teams.

Acknowledgments. This study was funded by Orange Research via the VauCanSSOn project and Télécom SudParis. The authors would like to thank Maxime Belair for his contribution and guidance during the VauCanSSOn project.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abbadini, M., Facchinetti, D., Oldani, G., Rossi, M., Paraboschi, S.: Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia. pp. 149–162 (2023)
2. Akasaka, K., Nakamura, A.: Reproducible software vulnerability testing with iac. In: 2020 International Conference on Computational Science and Computational Intelligence (CSCI). pp. 36–42 (2020)
3. Anwar, A., Abusnaina, A., Chen, S., Li, F., Mohaisen, D.: Cleaning the NVD: Comprehensive Quality Assessment, Improvements, and Analyses. IEEE Transactions on Dependable and Secure Computing **19**(6), 4255–4269 (2021)
4. Benedetti, G., Solarin, O., Miller, C., Tystahl, G., Enck, W., Kästner, C., Kapravelos, A., Merlo, A., Verderame, L.: An Empirical Study on Reproducible Packaging in Open-Source Ecosystems. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 1052–1063 (2025)
5. Bui, Q.C., Scandariato, R., Ferreyra, N.E.D.: Vul4j: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 464–468 (2022)
6. Challande, A., David, R., Renault, G.: Building a Commit-level Dataset of Real-world Vulnerabilities. In: CODASPY '22: Twelveth ACM Conference on Data and Application Security and Privacy, Baltimore, MD, USA. pp. 101–106 (2022)
7. Cosmo, R.D., Zacchiroli, S.: Software Heritage: Why and How to Preserve Software Source Code. In: Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017, Kyoto, Japan (2017)
8. Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., Wang, G.: Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In: 28th USENIX security symposium (USENIX Security 19). pp. 869–885 (2019)
9. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. MSR '20, ACM (Jun 2020). https://doi.org/10.1145/3379597.3387501
10. Householder, A.D., Chrabaszcz, J., Novelly, T., Warren, D., Spring, J.M.: Historical Analysis of Exploit Availability Timelines. In: 13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20) (2020)
11. Huang, M., Fan, W., Huang, W., Cheng, Y., Xiao, H.: Research on Building Exploitable Vulnerability Database for Cloud-Native App. In: 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (2020)
12. Kang, H.J., Nguyen, T.G., Le, B., Păsăreanu, C.S., Lo, D.: Test Mimicry to Assess the Exploitability of Library Vulnerabilities. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA (2022)
13. Kilian, S., Tong, V.V.T., Lalande, J., Majorczyk, F., Sanchez, A., Talon, N., Besson, P., Orsini, H., Lledo, P., Gimenez, P.: CasinoLimit: An Offensive Dataset Labeled with MITRE ATT&CK Techniques. In: 28th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2025, Gold Coast, Australia, October 19-22, 2025. pp. 523–537. IEEE (2025)
14. Li, X., Moreschini, S., Zhang, Z., Palomba, F., Taibi, D.: The Anatomy of a Vulnerability Database: A Systematic Mapping Study. Journal of Systems and Software **201**, 111679 (2023)

15. Meneely, A., Srinivasan, H., Musa, A., Tejeda, A.R., Mokary, M., Spates, B.: When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA. pp. 65–74 (2013)
16. Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., Wang, G.: Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In: 27th USENIX Security Symposium, Baltimore, MD, USA. pp. 919–936 (2018)
17. Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., Acar, Y.: VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA. pp. 426–437 (2015)
18. Reis, S., Abreu, R.: SECBENCH: A Database of Real Security Vulnerabilities. In: Proceedings of the International Workshop on Secure Software Engineering in DevOps and Agile Development co-located with ESORICS 2017, Oslo, Norway. CEUR Workshop Proceedings, vol. 1977, pp. 69–85 (2017)
19. Ruan, B., Liu, J., Zhang, C., Liang, Z.: KernJC: Automated Vulnerable Environment Generation for Linux Kernel Vulnerabilities. In: Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses. p. 384–402. RAID '24, New York, NY, USA (2024)
20. Schreuders, Z.C., Shaw, T., Shan-A.-Khuda, M., Ravichandran, G., Keighley, J., Ordean, M.: Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events. In: Gondree, M.A., Podhradsky, A.L. (eds.) 2017 USENIX Workshop on Advances in Security Education, ASE 2017, Vancouver, BC, Canada, August 15, 2017. USENIX Association (2017)
21. Shen, S., Kolluri, A., Dong, Z., Saxena, P., Roychoudhury, A.: Localizing Vulnerabilities Statistically From One Exploit. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. ASIA CCS '21 (2021)
22. Ullah, S., Balasubramanian, P., Guo, W., Burnett, A., Pearce, H., Kruegel, C., Vigna, G., Stringhini, G.: CVE-Genie: An LLM-Based Multi-Agent Framework for Reproducing CVEs. In: Proceedings of the 33rd ACM Conference on Computer and Communications Security. CCS '26 (2026)
23. Zerouali, A., Mens, T., Decan, A., González-Barahona, J.M., Robles, G.: A Multi-Dimensional Analysis of Technical Lag in Debian-based Docker Images. Empir. Softw. Eng. **26**(2), 19 (2021)

A Appendices

A.1 Package Selection Using Categories

Table 3 shows the breakdown of the packages we left out, and the proportion of vulnerabilities affecting each category, for the 2014-2023 period.

Even if the relevance criteria only filter out one third of the packages, it removes around half of the CVEs to consider, since web browsers (which are out of our scope) make up for a vast amount of vulnerabilities.

Table 3. Proportion of Debian packages affected by CVEs over the 2014-2023 period, sorted by categories (the ones filtered out w.r.t. our use case and the relevant remaining packages). It is worth noting a CVE can be counted twice, if it affects both a non-relevant package and a relevant one (e.g. CVE-2023-48795 affects Filezilla, a client app, and python-asyncssh, a library); that is why the first column does not add up.

Category	CVEs		Packages		Examples
Client Apps	3,059	(36%)	148	(16%)	browsers, GUIs
Developer tools	106	(1%)	28	(3%)	VCSes, compilers
Low-level	2,363	(19%)	117	(12%)	hypervisor, filesystems
Device config.	19	(0.1%)	5	(0.5%)	bluetooth, net. manager
Relevant	4,674	(44%)	653	(69%)	`tiff`, `sudo`, `php8.1`
Total CVEs	**9,978**		**951**		
Evaluated	**142**		**64**		

A.2 Another Example of a Fragile Exploitation

CVE-2021-4034: PwnKit. The `pkexec` application is a setuid tool designed to allow unprivileged users to run commands as privileged users according to predefined policies, and is similar to `sudo`. The vulnerable version of `pkexec` doesn't handle the command line arguments count correctly: when the argument vector is empty, the program ends up confusing environment variables and arguments. An attacker can leverage this to overwrite environment variables in such a way it will induce `pkexec` to execute arbitrary code, like a local privilege escalation given unprivileged users administrative rights on the target machine.

With a vulnerable version of `pkexec`, we can reproduce the CVE using publicly available exploits. However, following the PwnKit disclosure, a kernel patch was released 2 months later to block this kind of attack,[16] by introducing a safe guard in the `execve` syscall to add an empty argument when the `argv` vector is empty. This patch thwarts PwnKit completely. We thus need to use a previous version of the Linux kernel to reproduce the vulnerability, which we cannot handle easily using only Docker containers, since they share the same kernel as the host by construction. This creates an unexpected condition on the *kernel version* to reproduce a *pure userland* vulnerability.

[16] `https://github.com/torvalds/linux/commit/dcd46d897adb70d63e0`