Death Is Not the End: A Longitudinal Study on the Impact of Automatic Updates on Container Vulnerability Lifespans

Simge Tekin*, Octavian Suciu†, Sungsu Kwag*, Yonghwi Kwon*, and Tudor Dumitras*
*Department of Computer Science, University of Maryland, College Park, MD, USA
†Google Research, New York City, NY, USA
*{stekin, sskwag, yongkwon, tudor}@umd.edu †osuciu@google.com

***Abstract*—The emergence of immutable infrastructures such as container ecosystems has transformed how software is built, deployed, and maintained. In particular, patches that were traditionally delivered through in-place updates are now applied through image rebuilds and propagated through hierarchies of dependent images. Despite these substantial structural changes in patch delivery, the security impacts, such as the lifecycle of vulnerabilities and patching responsibilities in the software supply chain, remain understudied.**

Examining maintainer and user interactions across official Docker repositories, we find that maintainers often rely on automated patching of inherited vulnerabilities rather than intervening manually, while unclear maintenance timelines and responsibility boundaries impede remediation when automation halts. Building on these insights, we present the first longitudinal study of vulnerability lifespans in the Docker ecosystem, spanning six years. We analyze over 9,000 CVEs across 137 applications (from 756,313 images), and find that 78% of inherited vulnerabilities remain unresolved 30 days after disclosure. The predominant cause of prolonged exposure is the breakdown of automated patch propagation due to upstream end-of-life (EOL) events, leaving 11%—and up to 23% in deeper dependency layers—of inherited vulnerabilities unpatched even when fixes exist. Based on these findings, we provide actionable recommendations to reduce the window of exposure to vulnerability exploits and release a tool to improve transparency. More broadly, our work provides empirical insights into the fragility of automated patch propagation in software supply chains, emphasizing the need for clear maintenance practices and accountability across dependency hierarchies.

1. Introduction

Cloud computing has shifted the ways in which software is developed, delivered and deployed. One of the most notable shifts is the *automated and continuous patching of software vulnerabilities* [1], performed in containerized ecosystems such as Docker. Unlike in traditional deployments, where the OS and dependencies are updated in place, Docker images cannot be modified, as they are *immutable*. Consequently, all software updates (including vulnerability patches) are delivered through new versions of the images that traverse a standardized build–test–scan pipeline and are therefore expected to expedite updates [1], [2].

Containers mitigate well-known barriers to patching, such as concerns about incompatibilities [3] and the need to juggle multiple update managers [4], as patches are applied by rebuilding the entire image in a controlled and unified build process. They also reduce the fear of destabilizing software [5], as each application runs in an isolated container.

Container images have hierarchical structures: they are typically derived from a *base image* that provides the operating system or runtime, and are extended with other dependencies and application code. This structured composition makes the software supply chain explicit, providing *complete visibility* into all components—and therefore all known vulnerabilities—and supporting the generation of Software Bills of Materials (SBOM) [6], which incentivizes timely vulnerability remediation.

Automated patching applies to this hierarchy as well; when a base image is rebuilt with updated components, all derived images can automatically incorporate these updates during their own rebuilds, allowing patches to propagate from upstream to downstream. The visibility and automated patching together represent an important advance in supply-chain security. Understandably, the developers and maintainers of containerized applications have come to rely on the automated update mechanism. We therefore ask the question: *For how long do vulnerabilities remain unpatched in container ecosystems*?

Unfortunately, the prior research on container vulnerabilities [7], [8], [9], [10], [11], [12], [13] did not answer this question because it focused on the *prevalence* of vulnerabilities at *a snapshot* in time. What remains unclear is whether these vulnerabilities are actually resolved quickly, as one might expect given the increased visibility and automation.

If this is not the case, and vulnerabilities have a long *lifespan* in container images, this would introduce insidious threats. First, adversaries can craft targeted attacks against users of vulnerable Docker images. Second, vulnerabilities that remain unpatched for a long time represent targets of opportunity for attackers, and their lifespan was shown to be a *good predictor for exploitation* [14]. Not knowing *the root causes* of patching inefficiencies would make it difficult to determine who is responsible for addressing them, and whether they will even be patched. More broadly, the *impact*

of the developers' reliance on automated updates on the security of software supply chains remains an open question.

To understand this reliance, we start by conducting a qualitative analysis of GitHub issues related to update delays, from the Dockerfile repositories of three popular applications. We identify 54 issue threads discussing CVE vulnerabilities that persist in those images, and we systematically code them using qualitative research methods [15]. Our analysis reveals several causes for the vulnerability persistence, including the expectation that patches will arrive through automated base image updates (even when manual remediation is possible), confusion over patching responsibilities and a lack of clarity on when base images reach their *end-of-life* (EOL).

These insights motivate a large-scale longitudinal measurement of vulnerability lifespans on the Docker ecosystem. Specifically, we investigate the effectiveness of automated patch propagation in fixing vulnerabilities, factors that affect the likelihood of receiving automated patches, the availability of alternative patching strategies, and the impact of EOL events, at different layers in the base image chain, on vulnerability lifespans. We encounter two measurement challenges, not addressed in prior studies of vulnerability and patching lifecycles [4], [16], [17], [18], [19], [20]. First, because vulnerabilities are addressed by releasing new immutable images rather than updating versions in place, the beginning and end of a vulnerability's lifespan are unclear, since these changes do not occur cleanly across versions. Second, prior work utilized simple two-state (birth→death) survival models [4], [8], [21], [22], which can bias the results [23] when the events in the vulnerability lifecycle are not independent of one another. For example, after an EOL event in the base image, supply-chain vulnerabilities cease to receive automated updates, thus dramatically reducing the likelihood of the vulnerability remediation, which is *not captured* by the simple two-state models.

We address the first challenge by introducing the concept of *lineage*, a tag-based construct that captures the longitudinal evolution of an image. A vulnerability's lifespan is defined by the first and last images in the lineage that contain it. We further define significant events, notably the end-of-life for application and base lineages, as these mark the structural points where patch propagation halts and vulnerability lifespans diverge across derived images. We address the second challenge by adopting *multistate survival analysis* [24], which can account for inter-dependent lifecycle events.

With this novel vulnerability lifecycle model, we quantify vulnerability lifespans in Docker images published between January 2018 and July 2024 (6.5 years). Our dataset of 756,313 images from 137 applications enables reconstruction of the lifespans of 9,156 unique CVE vulnerabilities totaling 2 million lifecycle events across 21,598 Docker tags.

We find that base image EOL is a major barrier for automated updates: 11% (and up to 23% in deeper dependency layers) of inherited vulnerabilities remain unresolved after their base lineage reaches EOL. Patching is not faster after the release of PoC exploits for vulnerabilities: only 5.4% of vulnerabilities were addressed within 30 days of the PoC publication. Manual remediation remains rare and delayed: explicit updates address only 4.5% of inherited vulnerabilities, often months after their availability, and overwrites are similarly underutilized. Patch propagation is strongly affected by tag structure as measured by lineage relationships: highly specific tags are associated with up to 35% decreased risk of receiving timely automated updates. Lineage structure serves as a more reliable indicator of tag generality than semantic versioning, which is often inconsistently applied across applications.

We emphasize that these delays occur despite the fact that the automated patching mechanism works as intended. The new barriers to patching that we report in this paper are tied to the strengths of the containerized software development paradigm.

In summary, we make the following contributions:

- We combine qualitative insights from 54 GitHub issue threads with large-scale measurements to examine how automated patching behaves in practice, surfacing expectations, failure modes, and responsibility boundaries.
- We introduce the concept of lineage, which enables us to formalize the lifespan of vulnerabilities in containers.
- We curate a dataset[1] of over 750K images with vulnerability and lineage details, and systematically study update delays and vulnerability lifespans across 137 applications over 6 years.
- We provide tooling and actionable recommendations to improve the security of this ecosystem.

2. The Pitfalls of Automated Patching

Container images can be derived from other existing images (i.e., base images)[2] where an image can, in turn, serve as a base for other images, creating chains of multiple dependent images.[3] The content of each base image in the chain, including their vulnerabilities, can propagate to all descendant images. Such dependency chains have been identified as a major source of vulnerabilities in container ecosystems [12].

When a new image is released under the same base image **tag** (a mutable label pointing to an image), derived images realize automated patching by rebuilding against the updated base image (e.g., rebuilding `openjdk:8-jdk-slim`, which is derived from `debian:bullseye-slim`, when `debian:bullseye-slim` is updated with security patches). As such, the image updates and rebuilds play a key role in automated patching, where curated image programs such as Docker Official Images, Google-managed Base Images, and

1. The database is accessible at https://osf.io/943se/?view_only=bcc41282eab644f2b965f910571c314b

2. The Dockerfile, which specifies the commands to build the image, begins with a `FROM` statement specifying the base image, such as `FROM debian:bullseye-slim`,

3. For instance, a chain of `OS`→`Library`→`Application` such as `debian`→`openjdk`→`orientdb`.

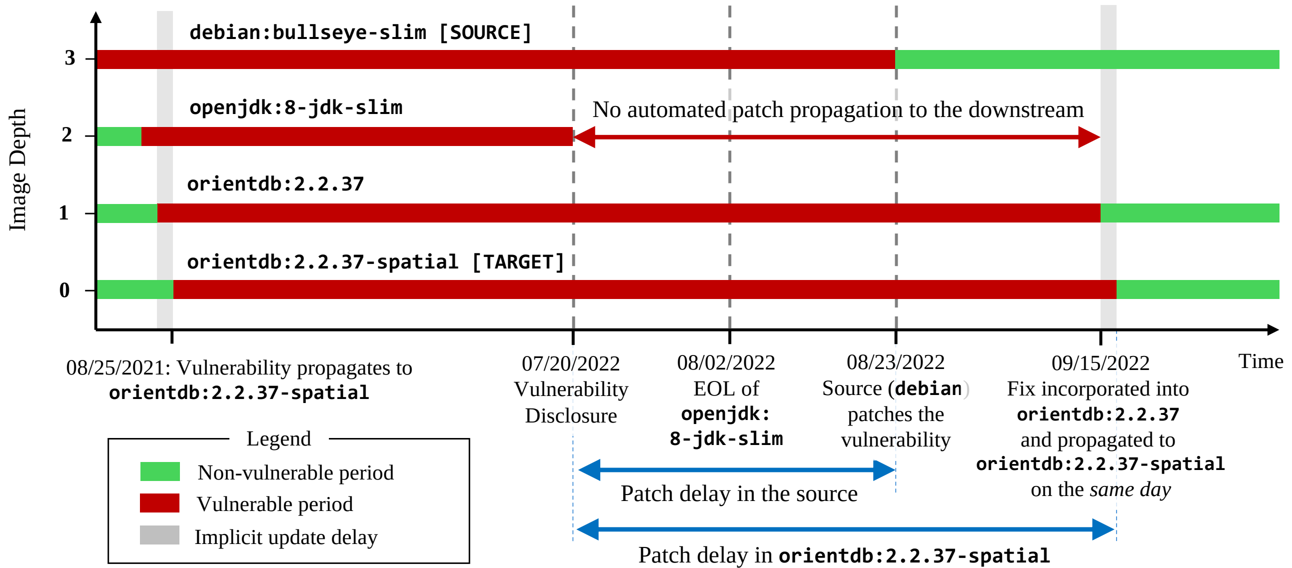


Figure 1: Lifespan of CVE-2021-46828, propagating through the base image chain (`debian:bullseye-slim` → `openjdk:8-jdk-slim` → `orientdb:2.2.37` → `orientdb:2.2.37-spatial`). An intermediate end-of-life (EOL) at depth 2 (`openjdk:8-jdk-slim`) interrupts patch flow, extending vulnerability exposure downstream.

Chainguard Images facilitate this process by regularly publishing updated images. These programs enable downstream developers, who build images on top of these curated bases, to benefit from automated patch propagation [25].

Security Expectations. Practices around container ecosystems, such as the maintenance effort for curated image programs, shape security expectations in container ecosystems. Specifically, images from curated image programs are regularly updated, and downstream developers expect those updates to be automatically propagated to derived images via image rebuild. This sentiment is also expressed in its documentation and security FAQ [26], which quotes three reasons why a CVE might remain visible in an official image: the upstream project has not released a fix (because the vulnerability is minor or the patch is unavailable); the listed CVE is a false positive; or the warning is a failure of the security scanner. Docker Official Images (ODI) program also operationalizes this expectation within its ecosystem: its continuous integration (CI) pipeline automatically rebuilds official images that are derived from other official images, ensuring that updates cascade within ODI.

2.1. Qualitative Analysis of Automated Patching

We conduct a qualitative analysis of maintainer and user interactions regarding patching behaviors to understand real-world practices around automated patching in container ecosystems and how these practices can create delays. In this study, users are often developers who build images based on existing ones, while maintainers are those responsible for publishing and updating their images.

Selection Criteria. We analyze GitHub issues under Dockerfile repositories of three popular Docker Official Images applications, Python, Node and Nginx [27], [28], [29] due to their real-world adoption (e.g., each application has millions of image pulls) and often being derived from a base image enabling us to effectively capture issues related to base image updates. We also include the central CI/CD metadata repository, official-images, which commonly receives issues about all official applications, similar to those in the application-specific repositories. To systematically identify relevant issues, we include all issues from the last five pages of each repository (i.e., last 125 issues), representing recent activity. We then manually filter out general support requests, troubleshooting discussions, and issues unrelated to security or update processes. Our final data set of 54 issue threads is composed exclusively of issues directly related to update delays, missing vulnerability fixes, and challenges related to base image or dependency updates. 46 issues specifically reference CVEs in the image, with 35 concerning vulnerabilities originating in the base image and 13 originating within the image itself.

Qualitative Coding Method. Two researchers independently open-coded these issues and their discussion threads to identify causes and patterns behind patching delays. After the initial coding, we synthesized a set of higher-level codes representing prominent themes. Each researcher then re-coded all threads using this codebook, and reconciled any differences through discussion to reach consensus on every code. Since our approach is entirely qualitative, we adhere to earlier research and do not report inter-rater reliability (IRR) [15], [30], [31].

2.1.1. Recurrent Themes. Our study reveals several recurring themes about developers' behaviors regarding patching in the container ecosystem.

Dependency on Base Image Updates. Maintainers try to stick with the automated patches: in 27 (out of 54) issues, maintainers explicitly stated that they were *waiting for a base image update* so the fix would propagate automatically via a rebuild rather than manually applying a patch (which is already available). For instance, a maintainer of `nginx` responded to a CVE report by explaining "*As the vulnerability itself is not in nginx - it will be fixed in due course as the fix is submitted into the base OS image* [32]."

Unclear Tag End-of-Life. A tag's end-of-life (EOL) is a concept unique to container systems: a tag reaches EOL when maintainers stop publishing updates under the tag, yet it remains accessible on container registries. In 6 issues, users reported problems with outdated tags that had already reached EOL, suggesting that they were unaware of the tags' maintenance status. In those cases, maintainers clarified that the tag will not be updated again as it is no longer supported [33], [34] or is deprecated [35].

Unclear Patching Responsibility. In 9 issues, maintainers and users were unclear about who is responsible for fixing certain vulnerabilities. Users reported issues, expecting maintainers to apply patches, while maintainers redirected them to base image repositories or external libraries. Moreover, in another 9 issues, although users asked maintainers to update images with available fixes, maintainers suggested manual workarounds (e.g., running "`apt-get upgrade`" in their own Dockerfiles), implying unclear responsibility boundaries.

2.2. Failures in Patch Propagation

We demonstrate how automated patching fails to meet expectations in practice by presenting the lifespan of a high-severity, remotely exploitable denial-of-service vulnerability, CVE-2021-46828.[4] The vulnerability originates from a low-level base image (`debian:bullseye-slim`), and propagates through the chain of images (`openjdk:8-jdk-slim` → `orientdb:2.2.37` → `orientdb:2.2.37-spatial`). The case is not a false positive (i.e., the vulnerable `libtirpc` package is indeed installed in the image) and the vulnerability was addressed in later releases of `libtirpc`.

Although the vulnerability was patched in the source image (`debian:bullseye-slim`) on 23 August 2022, the fix did not reach the end of the chain (`orientdb:2.2.37-spatial`) until 24 days later, when its immediate base image (`orientdb:2.2.37`) finally incorporated the patch.

Treating images as static artifacts, as prior studies have done, obscures the temporal dynamics of patching in container ecosystems. In practice, most vulnerabilities originate from the chain of base images [12], and maintainers commonly rely on upstream patch propagation to address them. Without a longitudinal perspective on how images evolve over time, propagation delays and failures, such as the one observed in the case of CVE-2021-46828 remain invisible.

4. in `libtirpc` prior to version 1.3.3rc1 [36]

Tag End-of-Life (EOL). A tag's EOL can be a threat to automated patch propagation. In the case of CVE-2021-46828 (illustrated in Figure 1), when `openjdk:8-jdk-slim` reached EOL, its derived image `orientdb:2.2.37` (and consequently `orientdb:2.2.37-spatial`) stopped receiving patches for 44 days, including the patch for CVE-2021-46828. The patch flow was restored when `orientdb:2.2.37` updated its base image to an actively maintained tag, `eclipse-temurin:8`.

The EOL events in traditional software product lines may also require major updates, but they occur infrequently (e.g., Nappa et al. found up to 17 lines across 10 Windows applications [4]), and are usually communicated clearly by the vendors. In container ecosystems, by contrast, each image tag behaves like a mini product line that is continuously updated, resulting in EOL events that occur more frequently. As a result, downstream users may experience different patching behaviors depending on the granularity of the tag they select. Worse, users have difficulty noticing an EOL event, partly because it occurs in subtle ways. For example, support for the `latest`[5] tag of the Rocky Linux container image (which previously tracked the version line 8) was discontinued on 07/07/2022, while the tag `8` continued to receive active maintenance.

Fragmented Responsibilities. The confusion about patch responsibility shows that users often treat images as black boxes and do not distinguish the source of vulnerabilities. This is partly because developers only control their immediate base image (i.e., the one specified in their Dockerfile), and the patch propagation directly depends on whether each image's immediate base incorporates the update. However, when a layer fails to patch, whether intentionally or due to oversight, responsibility for remediation becomes unclear. In Figure 1, `orientdb:2.2.37-spatial` was transitively affected when `openjdk:8-jdk-slim` reached EOL, even though it did not directly depend on `openjdk:8-jdk-slim`. The EOL event was hidden from `orientdb:2.2.37-spatial`, since it occurred one layer above its immediate base image in the dependency chain. This illustrates a structural issue: the deeper a vulnerability sits in the base-image chain, the more intermediaries a patch must traverse, increasing the likelihood of delayed propagation or unnoticed EOL events.

2.3. Systematic Study of Patching Pitfalls

Motivated by the observed reliance on automated updates and the pitfalls illustrated in Figure 1, we design a systematic measurement study asking the question: *How long do vulnerabilities remain unpatched in container ecosystems, and what factors influence these durations?*

To answer this, we account for the evolution of images over time, their base image chains, and the different update modes that fix vulnerabilities. We formally define the automated updates as: ***implicit updates***, where derived images

5. The default tag, which usually refers to the most recently built image for the version selected by the maintainers.

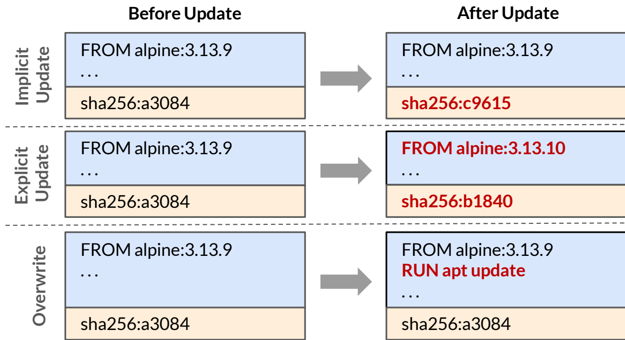


Figure 2: Three types of updates patching inherited vulnerabilities, with two consecutive images before and after each type of update. Blue and orange boxes denote Dockerfiles and base image digests, respectively. Updated parts are shown in bold.

receive updates by rebuilding against updated base images (the first row of Figure 2), without manual effort from downstream developers. We then examine the alternative mechanisms through which the supply chain vulnerabilities can be patched: (1) ***explicit updates***, where downstream developers manually migrate their base image references to newer versions (the second row of Figure 2), and (2) ***overwrites***, where the derived images independently fix the vulnerability (e.g., issuing an `apt update` command in their Dockerfile as in the third row of Figure 2), while the base image itself remains vulnerable.

We investigate the pitfalls of automated patching by quantitatively addressing the following research questions:

- ***RQ1***: *What is the effectiveness of automated patch propagation when fixing vulnerabilities?* (Section 4 and Section 5.2)
- ***RQ2a***: *How does the end of life of tags impact the lifespans of vulnerabilities?* (Section 5.2)
- ***RQ2b***: *What factors affect the likelihood of receiving automated patches?* (Section 5.3)
- ***RQ3a***: *To what extent do images independently patch inherited vulnerabilities (e.g., through overwrites)*? (Section 5.2)
- ***RQ3b***: *How does the depth of a vulnerability in the base image chain affect its lifespan?* (Section 5.2.1)

3. Measurement Methodology

3.1. Lineages

We introduce the concept of a lineage: the chronologically ordered set of images that a tag has referenced over time. Conceptually, this structure resembles a version control system: tags act like Git branches, while image digests correspond to commits. Each tag serves as a mutable reference that can be advanced to point to a newer digest as the image is rebuilt or patched, whereas digests themselves remain immutable records of past image states. The lineage thus captures how a tag evolves over time for software deployment, encompassing both the application and its entire runtime environment.

While no specific tagging scheme is enforced, the community encourages maintainers to adopt both general (e.g., `3` or `latest`) and specific (e.g., `3.13.9`) tags. This convention enables continuous patching through the mutability of general tags. For example, `alpine:3.13` may initially reference version 3.13.9 and later be updated to 3.13.10, whereas the specific tag `3.13.9` continues to reference the fixed 3.13.9 version.

Importantly, such tag updates are not limited to changes in the application version. Even if the application version remains constant (e.g., `alpine:3.13.9`), the underlying image may be rebuilt with updated dependencies. Over time, therefore, the same tag may reference different image digests, each incorporating newer security patches or system updates.

Lifecycle and End of Life. The *start date* of a lineage is the publication date of the first image that is pushed to a tag. The *end of life date* of a lineage is the publication date of the last image before the tag specifying the lineage is removed from the CI/CD metadata file (see Listing 1 for an example) that manages automated updates. Even after a lineage reaches end-of-life, the images remain publicly available.

Sub and Super Lineages. A single image may belong to multiple lineages (because an image can be referenced by multiple tags). We define super- and sub-lineage relationships based on image inclusion: a lineage is a super-lineage if it forms a *proper superset* of another lineage's images.

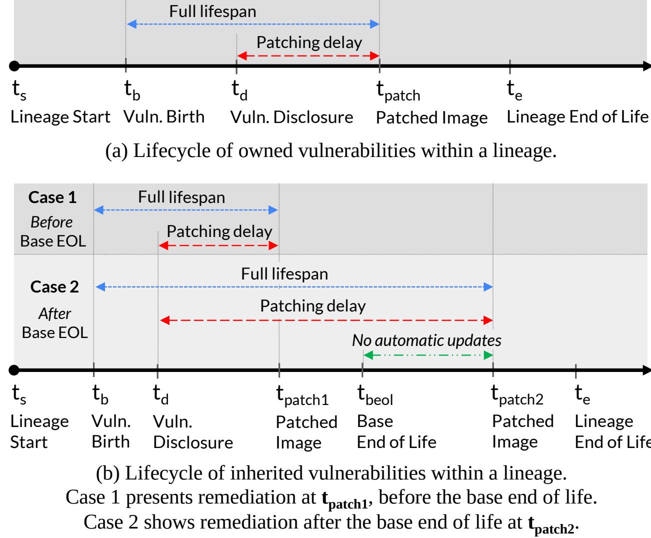
This typically occurs when generic tags (e.g., `debian:latest` are continuously updated to reference newer images that are also referenced by more specific tags (e.g., `debian:11.6` and `debian:11.7`).

We infer these relationships by comparing image sets across lineages, independently of the semantic version tags. A lineage may have multiple super-lineages; for example, `alpine:3.10.9`'s super lineages can be `alpine:3` and `3.10`. The number of super lineages allows us to systematically assess the *generality* of a lineage in the presence of inconsistent tagging practices across applications. Generic lineages include all the updates provided by their more specific sub-lineages, often have longer lifespans, and may implement more substantial changes.

Lineage Construction. We construct the lineages based on the tag history. Lineage construction is complicated by the fact that multiple images across different platform and architecture variants (e.g., `linux/amd64`, `linux/arm64`, and `windows/amd64`) share the same tag. These images may originate from the same Dockerfile or entirely separate ones, depending on the project's multi-platform build strategy. Additionally, tags may not be consistently managed across the variants. To this end, we restrict the lineages to contain images of *same platform* and *CPU architecture*, constructing the lineages of an application with the chronologically ordered set of images that are compatible with a specific architecture and platform. For example, Alpine

images (`3.13`, `linux`, `amd64`) and (`3.13`, `linux`, `s390x`) specify different lineages. We cover `amd64`, `arm` variants 5-8, `s390x`, `ppcle64`, `mips64le` and `386` variants of `linux` platform.

3.2. Vulnerability Lifespan



(a) Lifecycle of owned vulnerabilities within a lineage.

(b) Lifecycle of inherited vulnerabilities within a lineage.
Case 1 presents remediation at $\mathbf{t_{patch1}}$, before the base end of life.
Case 2 shows remediation after the base end of life at $\mathbf{t_{patch2}}$.

Figure 3: Lifecycle of owned and inherited vulnerabilities.

We categorize vulnerabilities based on their origins: (1) originating from the main application image and (2) inherited from the base image. Next, we explain the lifespan in relation to the update mechanisms explained in Section 2.3.

Owned Vulnerabilities. These vulnerabilities originate from the main image (i.e., the containerized application and its dependencies). Figure 3-(a) shows the events of the owned vulnerabilities' lifespan. t_b indicates the birth of the vulnerability (the publication date of first vulnerable image of a lineage) and t_{patch} represents the publication date of the first *non-vulnerable image*. The time window between t_b and t_{patch} represents the overall exposure of the lineage to the vulnerability. t_d marks a disclosure date of the vulnerability. Note that the birth event can also happen *after* t_d, if the vulnerability is introduced into the lineage after the disclosure (i.e., a vulnerable package is added after the disclosure).

A vulnerability is fixed by removing or updating the vulnerable package. While one can only update packages after the release of patches, removing vulnerable packages, which also eliminates the vulnerabilities, can be done at any point, meaning that t_{patch} can occur at any point after the t_b.

Inherited Vulnerabilities. Figure 3-(b) shows the lifespan of images originating from one of the images in base image chain. After a vulnerability is introduced into a lineage (t_b), it can be resolved through one of three mechanisms: implicit updates, explicit updates, or overwrites. Figure 3-(b) shows two cases of a vulnerability fix before and after the base end-of-life (t_{beol}).

- **Case 1:** If a vulnerability is fixed before the base EOL (t_{beol}), the lifespan and patching delay are similar to those of the owned vulnerabilities.
- **Case 2:** Once the base lineage reaches EOL (t_{beol}), implicit base image updates are no longer possible, and remediation must be handled by the downstream developers (i.e.,people who build on that image) through *explicit updates* or *overwrites*.

The interdependence of these events complicates the vulnerability lifespan analysis, leading to the analysis in Section 5.

3.3. Statistical Model for Vulenrability Lifespan

The use of inappropriate statistical techniques when computing complex lifecycle of container images can lead to misleading results. Below, we describe our statistical methodology and explain how it addresses potential biases.

Data Censoring and Truncation. There are vulnerabilities that have incomplete lifespans extending beyond our study period due to the longitudinal nature of our dataset. In statistical terms, these vulnerabilities are *right censored*. This does not mean that the vulnerability will not be patched, as the event may still occur in the future. Thus, treating these lifespans as complete results in an underestimation of their true lifespan.

The field of survival analysis [37], which is widely used in biostatistics, provides methods for handling time-to-event data (e.g., time to recovery/relapse for an ill patient) in the presence of right-censoring. These methods recognize that censored vulnerabilities may or may not be patched later; we only know they were unpatched up to a specific time. Another problem is *left truncation*, which occurs when subjects have contracted the illness before entering the study. In our dataset, this arises when a vulnerability is already present in the first observed image of a lineage that predates our dataset. For these vulnerabilities, the exact time they were introduced into the lineage is unknown, preventing us from measuring their age when they experience another event. To avoid bias from the uncertain introduction dates, we exclude these cases from our analysis, leveraging our sufficiently large dataset spanning an extended period.

Lifespan Definition. In our analysis, we envision *each vulnerability within a lineage* as a subject of study, analogous to a patient in biostatistics. We track vulnerabilities across chronologically ordered images in each lineage, similarly to sequential health checks in a longitudinal medical study, treating each image as a discrete observation point to systematically monitor vulnerability status over time. This mirrors the structure of longitudinal survival studies, where subjects are observed at successive time points to track the occurrence—or non-occurrence—of an event over time.

We consider the vulnerability lifespan as a random variable T. The survival function $S(t)$ captures the likelihood that the age of the vulnerability, from its introduction in the lineage to a "death event" (e.g., patching), is greater than t: $S(t) = \Pr[T > t] = 1 - F(t)$, where $F(t)$ is the cumulative

distribution function of the event happening before or at time t.

To avoid double-counting, we limit our analysis to the lineages without super-lineages, since sub-lineages replicate the image history of their supers. Note that the same vulnerability may appear in multiple independent lineages with distinct lifespans, and is treated as a separate subject in each case.

Multiple Events of Interest. Standard survival analysis focuses on scenarios where subjects can experience only one type of event. However, the lifecycle of both inherited and owned vulnerabilities in Docker are affected by multiple events. For example, an inherited vulnerability can be fixed in multiple ways (e.g., implicit updates that occur automatically or explicit updates that require manual action from the maintainers). Modeling these distinct update mechanisms as transitions into a single "vulnerability-death" state would mask their individual effect. Moreover, once a base lineage reaches EOL, inherited vulnerabilities from that base no longer receive automatic patches. Similarly, both inherited and owned vulnerabilities cannot be remediated when the main lineage reaches EOL.

In statistical terms, vulnerability remediation and EOL represent *competing risks*, and misclassifying such events is known to bias survival estimation [23]. In other words, when we do not observe the death event for a vulnerability in an EOL lineage, ignoring the fact that the EOL state drastically reduces the probability that the vulnerability will be patched in the future, can lead to underestimating its lifespan. We address this challenge by adopting the *multistate version* of survival analysis [24]. Specifically, we model the lifecycle of vulnerabilities by defining mutually exclusive states; a vulnerability transitions into a different state after an event of interest, which changes the likelihood of future state transitions. In the competing risks framework, the Cummulative Incidence Function (CIF) replaces the cummulative distribution function and captures the probability of experiencing the event of interest while not experiencing any blocking events by a specified time [38]. We estimate the CIF, the probability of being in a certain state at a given time with the non-parametric Aalen-Johansen estimator via `pymsm` [39]. This estimator is the multi-state generalization of the Kaplan-Meier estimator.

3.4. Dataset Description

We perform our analysis using the Official Docker Images (ODI) [40], a set of verified containers representing widely used software across operating systems (Alpine, Ubuntu, Debian), databases (MySQL, PostgreSQL, Redis), programming languages (Python, Node.js, Java/OpenJDK, Golang, PHP), web servers (Nginx, Apache), and other core services. Collectively, these images had over 154 billion pulls as of June 2025, reflecting broad real-world adoption.

ODI images are maintained by project developers and Docker engineers through standardized build pipelines that automatically rebuild images when their base images are updated, allowing us to directly observe automated patch propagation in practice. The ODI also provides rich historical metadata (image digests, architectures, layers), enabling construction of image lineages for longitudinal analysis. We include all 137 ODI applications as of April 2023, excluding Windows-based images since the vulnerability scanner used (Trivy) does not support them. Windows images represent only 1.7% of the total, so their exclusion does not significantly impact our results.

Historic Image Collection. The metadata archive of official images are hosted in the public GitHub repository 'repo-info' [41]. By traversing the commit history of repo-info, we retrieve historical image digests and use commit timestamps as the publication date of these images. From 137 official image repositories, we collect metadata of 762,858 images committed from January 2018 to July 2024. We set up a private Harbor container registry and pull the images by their digest IDs from Docker Hub to our registry.

3.5. Data Collection

3.5.1. Data Collection for Update Delays Analysis. We collect the release versions and timestamps of applications, and map each version to an image.

Version and Release Date Collection. We collect the release dates for each version of 97 (out of 137) applications included in the Docker Official Images program for which release dates are available. For 83 applications, we collect the release tags and their corresponding commits from their GitHub repositories, taking the commit dates as the release dates. For 14 applications, we manually collect the versions and release dates from their official websites.

Version to Image Mapping. For end-to-end update delay analysis, we map each project version to the first image published on Docker Hub. If the version of the application is specified in the image tag or tag aliases, we extract the versions from those tags. Then, we map the image digests to the version if there is a match. In total, we map 5,548 different version releases from 97 applications to images.

3.5.2. Data Collection for Lifespan Analysis. We scan the images we collected, identify base images, and analyze the lifespans of owned and inherited vulnerabilities using statistical methods.

Vulnerability Discovery. We use the open-source vulnerability scanner Trivy [42] to analyze the collected images. Trivy statically scans both operating system and language-specific packages and matches them against its local vulnerability database, which is regularly synchronized with multiple sources, including operating system and language vendor advisories, the GitHub Advisory Database, and the NVD. To ensure consistency, we normalize severity scores using the NVD database. Unlike scanners that inspect layers independently, Trivy scans merged images, ensuring that vulnerabilities removed by subsequent layers are not reported. By comparing the results of each image with those of its base image, we identify inherited vulnerabilities and those overwritten by child layers. Importantly, Trivy accounts

for backported vulnerabilities through vendor-specific advisories that reflect backporting practices, minimizing false positives by recognizing when fixes are applied without version changes [43].

Base Image Discovery. We determine the exact base image of each Docker image by leveraging layer information from the repo-info repository, following the approach in [7]. Since all layers of a base image are retained as the deepest layers in descendant images, we construct an image dependency graph (where vertices represent layers and edges denote inter-layer relationships) and label the topmost layer with the image's digest ID. Traversing this graph allows us to recover the full base image chain for each image.

However, identifying exact base lineages (tags) requires further steps, as a single image can be referenced by multiple tags. To address this, we link each image digest to its originating Dockerfile and extract the base tag from the FROM statement. This process is complicated by implicit updates and multi-architecture builds, which can generate multiple images from the same Dockerfile. We resolve this by parsing CI/CD metadata from the "official-images" repository [40], mapping tags, architectures, and applications to Dockerfiles, and associating the earliest image built after each metadata commit with the corresponding Dockerfile. This approach enables accurate tracking of base lineages.

4. Update Delays in Docker Images

We investigate how quickly the updates propagate in the Docker ecosystem. Specifically, we adopt the perspective of image maintainers, who aim to provide up-to-date software for containerized infrastructures. We do not distinguish between security fixes and functional updates. Instead, we aim to illuminate the timeliness of the "dockerization" process, after the code is written and released.

4.1. Delays in Application Updates

When a new application version or patch is released, maintainers update the Dockerfile, which triggers an automated rebuild and publishes a new image to Docker Hub, effectively delivering the patch for that containerized application.

We measure the end-to-end delay from an application's upstream release to the publication of its corresponding Docker image. Across 5,548 application versions from 97 official images, the mean delay is 5.7 days (median: 2.3), indicating efficient propagation from the upstream application into its containerized form. However, a small fraction (about 1%) of updates experience delays exceeding a month. For some of these cases, we find related GitHub issues under the official-images repository mentioning the build failures that can be attributed to the delay. These failures typically occur when images for certain tags or architectures cannot be built successfully, often due to incompatibilities or infrastructure-specific issues and are typically reported by upstream maintainers or users [44], [45], [46].

In addition, these build failures cascade, causing downstream builds to fail. For example, the failure to build `alpine:3.14` for the `s390x` architecture delayed the release of derived images such as `php:7.4.21-fpm-alpine3.14` [47] and `node:current-alpine3.14` [48] by 50 days. Such build failures introduce substantial variance in update delays and represent a hidden source of latency.

4.2. Delays in Supply Chain Updates

We measure the delay in updating base images within derived images by analyzing 21,598 lineages spanning 137 applications and 739,159 images.

Explicit Supply Chain Update Delays. The delay is measured by the time difference between the start date of the base lineage and the publication date of the derived image. It is essentially the delay since the earliest date the base tag update could have been applied. We identify 20,461 explicit updates in total, meaning that most lineages experience less than 1 explicit update throughout the lineage lifespan.

For delay computation, we include only updates to a different tag within the same project, excluding major base changes (e.g., `ubuntu` to `debian`) to focus specifically on version updates rather than system shifts. The average delay for explicit updates is 242 days (median: 74 days), indicating that explicit updates often lag several months behind the availability of new base tags. Notably, 774 of these explicit updates are later downgraded back to the previous base tag. A manual inspection of these cases reveals maintainer comments on GitHub, indicating that some of these reverts are caused by *compatibility issues* between the application and the updated base image [49]. This illustrates a key challenge of explicit updates: even when a newer base image is available, maintainers may be unable to adopt it due to breaking changes.

Implicit Supply Chain Update Delays. We identify 640,527 implicit updates, making them far more common than explicit updates. Moreover, implicit updates propagate significantly faster, typically within 3.7 days on average (median: 1 day) after the upstream base image is published.

Our analysis reveals a striking disparity between the supply chain update mechanisms. In contrast to the fast propagating implicit supply chain updates, explicit base image updates are both rare and slow illustrating the efficiency of automation in the Docker ecosystem. As a result, the vast majority of timely patch propagation in Docker images depends on upstream base images being updated and rebuilt. In particular, any lag or discontinuation in base image maintenance can lead to widespread, persistent vulnerability exposure downstream, making base image maintenance a critical point of ecosystem.

5. Impact of Container Patching Delays

Known software vulnerabilities may remain unpatched for extended periods, affecting various stakeholders. In particular, developers who use Docker images would be con-

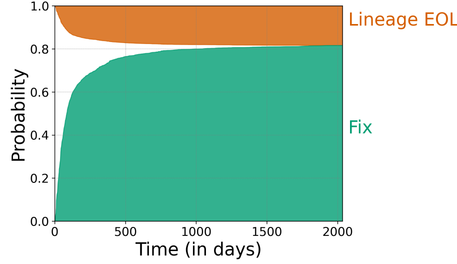


Figure 4: Cumulative incidence functions (CIF) of vulnerability patching and lineage end of life (EOL) events, measured post-disclosure.

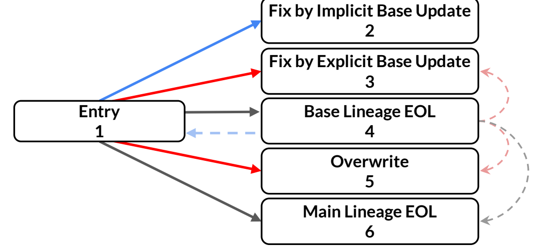


Figure 5: State diagram of lifespan of inherited vulnerabilities. Red arrows depict the updates by an action performed by main lineage maintainers. Blue arrow depicts the patches propagated implicitly from the supply chain.

cerned about the vulnerabilities and their lifecycles within the containers. To this end, we answer the following questions from the perspective of developers: *For how long will my infrastructure remain exposed to these vulnerabilities?* and *What factors influence the lifespan of supply-chain vulnerabilities?*

We extract 2,938,448 lifespan observations spanning 9,156 unique CVEs (41% being High or Critical according to NVD severity rankings [50]) and 21,598 lineages across 137 official image applications. Of these, 1,451,689 observations correspond to owned vulnerabilities originating in the application images, while 1,486,759 represent inherited vulnerabilities originating from their upstream base images. To avoid overrepresenting applications with large numbers of vulnerabilities, and to better reflect real-world impact, we draw a sample of 500K observations for each of owned and inherited vulnerability analyses, proportional to the square root of each application's pull count.

5.1. Lifespan of Owned Vulnerabilities

The state diagram consists of three states for owned vulnerabilities: Entry, Lineage End of Life, and Fix. When computing the post-disclosure lifespan (i.e., patching delay in Figure 3-(a)), the entry date is defined as the later of the first appearance of the vulnerability in the image lineage (t_b) and the public disclosure date of the vulnerability (t_d). If the lineage was already vulnerable before disclosure (i.e., $t_b < t_d$), the period between [t_b, t_d] represents the zero-day exposure window [51].

From the entry state, a vulnerability can transition to "Fix" when the first non-vulnerable image is published (t_{patch}), to "Lineage End of Life" (Lineage EOL) if the lineage becomes unsupported while still vulnerable, or remain censored if neither event occurs within the observation period. An example case for the owned vulnerability lifespan can be found in Appendix Section A.2.

Figure 4 shows how the estimated proportion of fixed vulnerabilities changes over time for post-disclosure lifespans, competing with the case of a lineage reaching its EOL.

Window of exposure for owned vulnerabilities. By day 30 after disclosure (the time frame recommended by CISA to fix high-severity vulnerabilities [52]), the probability of death for a vulnerability before lineage EOL is only 22%. Notably, many vulnerabilities had already been present in affected images for an extended period before disclosure, with a median pre-disclosure presence of 298 days (mean: 221 days), indicating substantial zero-day exposure before their public disclosure.

Upper bound for patch propagation to derived images. By the end of our measurement period, 81% of vulnerabilities had been fixed before their lineage reached EOL, leaving nearly 19% unresolved. This indicates that, under current ecosystem practices, the upper bound of patch propagation through automated (implicit) updates is around 81%. Vulnerabilities that remain beyond this threshold are unlikely to be resolved in derived images without explicit maintainer intervention.

5.2. Lifespan of Inherited Vulnerabilities

We model the events in the inherited vulnerability lifespan (see Figure 3-(b)) using the multi-state diagram in Figure 5. As with owned vulnerabilities, we present the post-disclosure lifespan analysis, based on entry at $\max(t_d, t_b)$, respectively. We apply this model uniformly across vulnerabilities regardless of the depth of their source in the chain of base images, in order to obtain an overall view of inherited vulnerability lifecycles.

The three primary remediation mechanisms, namely implicit updates, explicit updates, and overwrites correspond to transitions $1 \rightarrow 2$, $1 \rightarrow 3$, and $1 \rightarrow 5$, respectively, marking the time until t_{patch1} in Figure 3-(b).

Black arrows in the state diagram indicate events that preclude further remediation. The first such event is the EOL of the main image lineage ($1 \rightarrow 6$), after which no additional images will be built, and thus, the remediation is blocked. The second event is the EOL of the base lineage ($1 \rightarrow 4$), which prevents further implicit updates while allowing explicit updates or overwrites (depicted as t_{patch2}).

After reaching the 4th state (Base Lineage EOL), three potential paths exist: (1) explicit updates may remediate some vulnerabilities, completing their lifespan ($4 \rightarrow 3$), or revert to the entry state ($4 \rightarrow 1$) if the new base lineage is still active; (2) vulnerabilities may be fixed through an overwrite ($4 \rightarrow 5$); or (3) they may persist until the main lineage

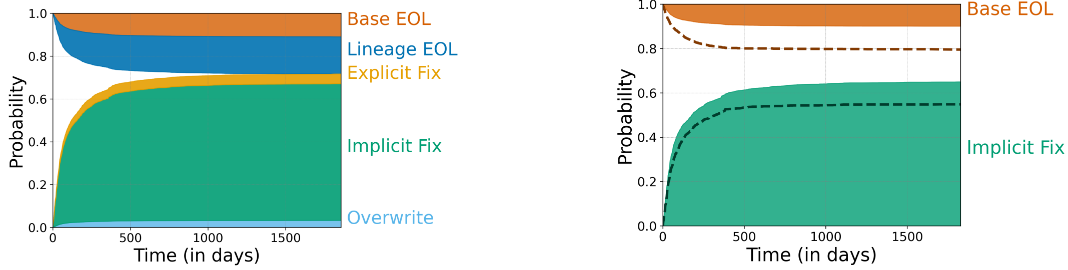


Figure 6: Cumulative incidence function (CIF) for events in vulnerability lifespan, throughout the 6-year, 7-month observation period. (Left: (a) the CIF for all inherited vulnerabilities. Right: (b) CIF stratified by vulnerability origin: depth 1 (i.e., immediate base image; filled curve) versus depth ≥ 3 (dashed curve) in the base image chain.

EOL ($4 \rightarrow 6$). In practice, less than 1% of vulnerabilities experience an overwrite or explicit update after reaching state 4.

Although the dashed transitions from state 4 mean the state diagram does not strictly meet the criteria for a classic competing risks model (due to possible state revisits), nearly all subjects reaching state 4 proceed directly to the terminal main lineage EOL (state 6) without further events. This implies that t_{patch2} usually does not occur after t_{beol} extending the vulnerability lifespan up to lineage end of life at t_e. Therefore, we pragmatically treat the Base Lineage EOL as terminal and exclude outgoing transitions from state 4 in our analysis.

Effectiveness of Update Mechanisms. The three remediation methods (i.e., implicit updates, explicit updates and overwrites) correspond respectively to transitions $1 \rightarrow 2$, $1 \rightarrow 3$, and $1 \rightarrow 5$, occurring by time t_{patch1} as shown in Figure 3-(b). Implicit updates, driven by upstream changes, covered the majority of remediations (63.8%), highlighting containers' reliance on supply chain updates. Explicit updates and overwrites, which require downstream developer actions, contributed significantly less (explicit updates: 4.5%, overwrites: 3.3%), reflecting limited intervention.

Window of Exposure for Inherited Vulnerabilities. The cumulative incidence function (CIF) in Figure 6-(b) highlights long-term exposure trends. By the end of the 6-year and 7-month observation window, 28.4% of post-disclosure vulnerabilities remained unresolved. Critically, at day 30 post-disclosure, a critical period recommended by CISA for addressing high-severity vulnerabilities, only 22% were remediated. Before their public disclosure, vulnerabilities had already existed in their respective lineages for a median of 256 days.

We also identify 184 distinct CVEs corresponding to 70,002 lifespan observations with published PoC exploits [53]. The result shows that only 5.4% of the vulnerabilities with PoCs were resolved within 30 days, implying that they are not specially treated despite the published PoCs.

Impact of Base EOL. 11% of vulnerabilities remained unpatched due to the base lineage EOL, representing scenarios of structurally blocked patch propagation. This underscores a lack of downstream intervention, particularly the use of explicit updates, that could mitigate the effects of base lineage EOL and restore the implicit patch flow.

Additionally, we present the lifespan after reaching state 4, as the *window without automatic updates* because the implicit updates are not possible during this period of time. Vulnerabilities that enter state 4 remain in the lineage for an average of 303 additional days (median: 168), typically persisting until the lineage itself reaches EOL. For instance, CVE-2022-23218 [54], inherited by `swift:centos7-slim` from `centos:7` on August 11, 2020, persisted for 441 days post-disclosure due to the base's earlier EOL on September 15, 2021, well before the project's deprecation notice on September 29, 2022. Overall, deprecation of `centos:7` affected 69 Docker tags across 7 applications in our dataset, while these application images were actively rebuilt and used for an average of 187 days, making them unable to receive patches for vulnerabilities in the base (see Appendix A.1).

Overwrites and Their Impact. Some vulnerabilities present in base images are overwritten immediately at the time of inheritance. For these vulnerabilities, t_b and t_{patch1} are equal. We do not include these vulnerabilities in our primary analysis, as they have never actually been inherited in practice. However, this is a prevalent phenomenon: including these vulnerabilities increases the overwrite ratio from 3.3% to 15%, highlighting that overwriting base image vulnerabilities during image construction is a significant mechanism for vulnerability resolution. Mechanisms such as multi-stage builds help exclude unnecessary base image components from the final image, reducing the number of inherited vulnerabilities.

Temporal Patterns. To assess the evolution of patching trends, we divide vulnerabilities into five yearly cohorts based on their entry year and repeat the same analysis, censoring each vulnerability after one year. The share of vulnerabilities remediated via implicit updates rose from about 56% in 2021–2022 to 64% in 2023–2024. This improvement likely reflects ecosystem-wide changes, including the integration of automated vulnerability scanners (e.g., Docker/Snyk) [55], heightened maintainer security awareness (e.g., Docker security announcements), and regulatory pressures such as SBOM requirements. Despite this progress, challenges persist: the share of vulnerabilities patched within

30 days after disclosure fluctuated between 15–22% over the six-year period. Note that because we censor at one year, the impact of long-tail events may be understated.

5.2.1. Effect of Vulnerability Depth. To examine how vague patch ownership across the image hierarchy affects the likelihood of receiving an automated update at different depths, we draw a stratified sample of 20,000 inherited vulnerability observations from each depth group: depth 1, depth 2, and depth ≥3. As shown on Figure 6-(b), as more intermediaries are added to the base-image chain, the probability of patch-propagation failures increases: our analysis shows that the probability of being stuck in the base-EOL state rises from 10% (depth 1) to 14% (depth 2) and 23% (depth≥3). This trend is mirrored in the likelihood of vulnerabilities being addressed through implicit updates, which decreases from 65% at depth 1 to 62% at depth 2 and 55% at depth≥3.

Among all inherited vulnerabilities in our dataset, 81.8% originate at depth 1 (i.e., directly inherited from the immediate base image), 11.6% occur at depth 2 (i.e., inherited from the base image of the immediate base image), and 6.6% appear at depth ≥3 of their chain of base images. These proportions reflect the fact that many official images have relatively short base image chains since they usually serve as bases for downstream applications. Depths in our dataset ranges up to 6.

5.3. Factors Affecting Automated Patch Delivery

To understand how specific features of lineages, vulnerabilities, and developer communities affect the automated patching of inherited supply chain vulnerabilities after public disclosure, we employ Cox regression to estimate the hazard ratio (HR) [56]. The hazard ratio quantifies the relative instantaneous rate at which an event (e.g., vulnerability remediation) occurs in one group compared to a reference group, conditional on not yet having occured. In our context, '$HR > 1$' indicates that remediation is more likely to occur sooner for the group of interest relative to the reference group, whereas '$HR < 1$' means a slower remediation rate. Table 1 summarizes the HRs for the covariates in our model (see Table 3 for full table in Appendix). Notably, features related to the base lineage influence remediation rates and provide actionable guidance in selecting base images for faster resolution of vulnerabilities.

Table 1: Hazard Ratios for Implicit Fixes (N=1,000,000)

Covariates	Units / Comparison	HR
Base maintainer is Docker community	Docker vs upstream	0.44
Repo maintainer is Docker community	Docker vs upstream	0.97
Number of super lineages of the base	1 more super	0.94
Repo pull count: High	vs Low	0.90
Base repo pull count: High	vs Low	1.23
Arch = s390x	vs amd64	1.20
Base severity = CRITICAL	vs MEDIUM	1.25

Tag Generality: Super Lineage Count. To represent the generality of a lineage (or a tag), we introduce the super lineage count, which is the *number of super lineages*. Note that we do not use the semantic versioning level of a tag (e.g., "major" vs. "minor") due to the inconsistent tagging practices: many projects maintain only certain tag types (e.g. `elasticsearch:8.17.9`) or do not adhere to semantic conventions (such as `nginx:mainline` or `debian:bookworm`).

In our analysis, each additional super lineage is associated with a 6% decrease in remediation hazard by an implicit update ($HR = 0.94$). Intuitively, additional super-lineages mean a more specific tag (or less generic tag), which has a lower probability of receiving timely automated patches.

For example, if a base lineage has 7 super lineages (the maximum observed in our dataset), the cumulative effect on the hazard is: $HR = 0.94^7 \approx 0.65$, meaning that the remediation rate is reduced by approximately 35% from a vulnerability inherited from a top-level lineage (with no super lineages).

Maintainer Community. The hazard of remediation for inherited vulnerabilities from base images maintained by the Docker Community (i.e., Docker engineers and community members) are 56% *less* compared to those inherited from images maintained by upstream communities. This indicates these applications are slower to patch their own vulnerabilities, which in turn delays the propagation of automatic updates to downstream images. In contrast, upstream project communities (e.g., the NGINX Docker Maintainers) likely benefit from deeper domain knowledge, enabling them to integrate fixes more promptly. On the other hand, if the affected application itself is maintained by Docker, the remediation hazard is only slightly lower ($HR = 0.97$), suggesting that the main bottleneck lies in the slow fixes at the level of Docker-maintained base images themselves.

Popularity. Based on the total pull counts, each application can have three levels of popularity: Low, Moderate, or High. Faster implicit patching is linked to the use of popular base images, not the popularity of the application itself.

Architectures. `amd64` is the most widely used and serves as the default architecture. However, other architectures (`s390x`, `386`, `ppc64le`, `arm`, and `mips64le`) see faster remediation compared to `amd64` (HRs 1.12–1.19).

Vulnerability Severity. To assess the role of severity in patch prioritization, we use CVSS v3/v3.1 base scores, which combine exploitability and impact subscores. High-severity vulnerabilities are fixed more quickly than medium ones ($HR = 1.25$), in contrast to low-severity vulnerabilities ($HR = 0.58$). Complementing our regression analysis, classical survival analysis (treating vulnerabilities as either fixed or censored) shows a median patch time of 134.5 days for critical vulnerabilities, 247.4 days for high, and 625 days for low severity.

5.4. Unofficial Images

Our lineage-based abstractions are relevant to many non-official images, as it is common practice to push updates to tags across both official and unofficial repositories. The update types we observed also apply broadly.

Table 2: Update Frequency and Vulnerability Metrics

Metric	Official		Unofficial	
	Mean	Median	Mean	Median
Update frequency (per month)	2.5	2.0	1.1	1.0
Vulnerability resolution rate	0.054	0.02	0.110	0.02
Vulnerability volume (CVEs)	418	107	1446	703

Updating and Patching Behaviors. To understand whether our insights and findings are applicable to unofficial images, we perform a comparative analysis of 16 popular unofficial Docker applications including programming languages (e.g., Golang), databases (e.g., PostgreSQ), web servers (e.g., Nginx), and data platforms, each with millions of pulls. We pulled and scanned these unofficial image tags daily for one month (Jun–Jul 2025) to build a historical archive of updates and vulnerabilities, this historical information is only available for the official images.

For each application, we analyze the lineage defined by '`linux/amd64`' and the latest tag (when available), or the most recent tag of the major version at the time of our analysis. For each unofficial image, we identify a corresponding official image and apply the same methodology, enabling direct comparisons of update frequency, vulnerability volume, and resolution rates. We assess the differences between official and unofficial images with respect to: (1) update frequency (i.e., how often each tag was updated to reference a new image): (2) vulnerability resolution rate (i.e., the proportion of CVEs fixed relative to the number observed) ; and (3) vulnerability volume (i.e., the total number of unique CVEs present in each image's lineage).

The results shown in Table 2 suggest that unofficial images tend to accumulate a substantially larger volume of vulnerabilities over time and are updated less frequently compared to their official counterparts. Although unofficial images exhibited a slightly higher vulnerability resolution rate during the observation period, their overall backlog of unresolved CVEs remains much higher. This indicates that, in practice, official images present a smaller attack surface.

We also observe that unlike official images, it is not standard for unofficial images to be rebuilt immediately following a base image update. Many popular unofficial applications are rebuilt periodically or manually to incorporate upstream changes, which often delays the adoption of base image security fixes in downstream images.

Threats to Validity. The results are subject to several limitations. First, the results may vary depending on the policies of the maintainer community of each application and their build infrastructure. Second, our comparative analysis covers a selected sample of popular images and a one-month observation period; as such, it may not fully capture the diversity or long-term trends present in the broader Docker ecosystem.

6. Discussion

Security Impact of EOL Base Lineages. We observe that inherited vulnerabilities tend to have long lifespans, highlighting the challenge of securing the software supply chain in immutable infrastructures.

The root cause is the widespread use of base lineages that have reached EOL. Our qualitative analysis also shows that users often lack clear information about which tags are EOL, especially when EOL events occur in deeper layers of the base image chain, contributing to the continued use of unsupported images. Note that the EOL phenomenon is not unique to containerized applications. Providers of non-containerized software may also maintain multiple product lines, which stop receiving updates after their respective EOLs [4]. However, a Docker image may select any version tag for its base, no matter how fine-grained, making EOL events more frequent. Combined with the reliance on implicit updates, the widespread use of specific, fine-grained version tags means that EOL events have a disproportionately large impact on how long vulnerabilities persist. Therefore, we recommend integrating EOL tracking mechanisms directly into CI/CD pipelines, along with suggestions for replacement lineages that continue to receive implicit updates, to provide this information automatically and consistently, rather than requiring application image maintainers to obtain it through ad-hoc channels. To facilitate the development of such a measure, we release our tool that clearly indicates the EOL status of lineages. The tool can be found at tool.vulineage.com.

A More Reliable Lineage Versioning Scheme. We observe an *inconsistent* application of semantic versioning tags, such as major and minor version tags, across various applications. In addition, it is often unclear what versions the commonly used tags such as `latest`, `stable`, or `mainline` actually track, a challenge also reported in prior studies [7].

These inconsistencies often lead to confusion and make it difficult for developers to select appropriate base lineages. Similar problems with semantic versioning practices were also found to discourage updates of third-party libraries in Android [3]. In the Docker ecosystem, unreliable semantic tags make it difficult for developers to predict which base lineages are likely to reach EOL. Based on our analysis, we recommend that container developers select base lineages according to their sub-super lineage relationships rather than relying solely on semantic tags. Our results indicate that this structural relationship is a more reliable indicator of which lineages will continue to receive implicit updates. To support this best practice, we release a tool that visualizes sub-super relationships among lineages, making it easier for developers to understand tag relationships and to select and track robust base images.

Integration of Comprehensive Tests. Updating base images, whether a minor patch or a major upgrade, can be challenging for developers due to integration complexities between the base image and the main application. Our analysis shows that explicit updates are both rare and delayed, even though they are essential for restoring the implicit patch flow once it breaks. Moreover, about 4% of explicit updates are later reverted, often due to functionality breakage. Unfortunately, we observe that the current Docker images are only tested briefly (e.g., running simple build status checks).

There is no comprehensive functionality testing integrated into CI/CD pipelines. We recommend integrating unit and regression testing as a part of the CI/CD pipeline to mitigate such reverted updates, and make it easier to apply explicit updates. This could also encourage developers to select more generic base lineages—which are likely to implement more substantial changes—thus further reducing the lifespan of inherited vulnerabilities.

Under-Utilized Vulnerability Remediation Mechanism. Our analysis reveals that overwriting vulnerabilities inherited from base images is an effective but underutilized remediation strategy. Immediate overwriting at the point of inheritance is common enough to raise the overall overwrite ratio to 15%; however, overwriting is rarely used after a vulnerability has been inherited into the main image, with only 3.3% of supply-chain vulnerabilities remediated in this way. We also observe this pattern in our qualitative analysis where maintainers frequently suggest manual overwrites as a workaround for downstream users, but rarely apply these fixes in the official images themselves.

Given the findings, our recommendation on overwriting is to use it as a targeted downstream remediation mechanism *only under limited conditions*: when a base lineage has reached EOL and migration to a maintained base is infeasible, or when upstream remediation cannot meet the required security timeframe. This restriction is necessary because overwrites may introduce compatibility issues by replacing individual packages outside the dependency set originally selected and tested in the base image, potentially causing version mismatches or behavioral incompatibilities. Moreover, overwriting is inherently unscalable, as it does not address the vulnerability at its source in the base image. Hence, its use should be narrowly scoped (e.g., limited to critical security updates) and accompanied by extensive testing to mitigate compatibility and regression risk.

Base Image Selection Strategies. As discussed above, selecting base lineages with fewer super lineages significantly reduces the survival of inherited vulnerabilities. Repository popularity also plays a key role. Popular repositories tend to provide faster vulnerability fixes downstream.

Exploitability in Practice. Although the practical exploitability of the long-lived vulnerabilities detected in our study depends on deployment context, the presence of a known-vulnerable component in a container image remains security-relevant. This is because even if the primary application does not use a vulnerable package, it may still become reachable through derived images, configuration, debugging, or backup procedures. For example, a Debian-based PostgreSQL image may not invoke an inherited `xz` binary during routine database service, yet a vulnerability such as CVE-2024-3094 [57] in `xz/liblzma` remains relevant if operators later use it inside the container for backup compression. We therefore study vulnerability persistence from a delivery-stage supply-chain perspective, treating the inclusion of known-vulnerable components in distributed images as a meaningful security concern, even when exploitability is deployment-dependent.

7. Related Work

Prior work has explored security facets of containers: container attack surfaces [58], [59], secret leaks [60], and misconfigurations [61]. Shu et al. [7] studied the prevalence of vulnerabilities in containers. Zerouali et al. [8] analyzed the link between outdated packages and vulnerabilities in containers. Lu [9] studied vulnerability distribution across repositories. Wist et al. [10] examined vulnerability characteristics and image update frequency, while Zerouali et al. [11] measured patched vulnerabilities in Debian-based images. However, these snapshot-based analyses do not track how vulnerabilities evolve over time or how long they remain unpatched. To gain a more comprehensive understanding of the container ecosystem's patching mechanisms, it is necessary to consider the lifespan of vulnerabilities. The closest longitudinal study, Liu et al. [13], manually analyzed 334 vulnerabilities across 30 images, but still relied on a single snapshot of Docker Hub. Their approach inferred vulnerability lifespans by ordering tags chronologically, yet this is inherently limited: patches may be introduced in earlier images within the same tag, and tags themselves are not guaranteed to be semantically consistent or comparable.

In the context of software supply chains, prior work mapped open-source attack surfaces [62] and measured the malicious package prevalence in popular ecosystems [63]. Cito et al. [64] analyzed Dockerfiles and identified missing version pinning as a major barrier to reproducible builds. Base images, central to the container supply chain, have been studied for inherited vulnerability exploitability [12] and propagation [7]. Other studies analyzed deviations from best practices (e.g., limited permissions, non-default passwords) and the characteristics and evolution of the images. [65], [66]. We present a comprehensive model analyzing the impact of automated patching mechanisms on inherited vulnerability lifecycles and identifying challenges in patch delivery from a software development perspective. Patching delays in non-containerized software have been extensively studied, with prior work attributing delays in closed-source applications to human factors and technical challenges [16], [67], [68]. In open-source ecosystems, studies have analyzed the full vulnerability lifecycle [19], [20], and Nappa et al. [4] applied statistical techniques from medical research to assess real-world patch deployment dynamics.

Extending on this line of research, researchers have qualitatively analyzed GitHub issues to identify information types [69] and GitHub contributions to expose the risk of contribution abuse [70]; we adopt a similar qualitative approach to identify reasons of patching delays analyzing GitHub issues.

8. Conclusions

We present the first longitudinal mixed-methods study of vulnerability lifecycles in immutable Docker container ecosystems, tracking over 9,000 CVEs across 137 official images over six years.

Our lineage-based model and multistate survival analysis reveal that (1) base lineage end-of-life (EOL) emerges as a key structural choke point in automated patch propagation: 11% of inherited vulnerabilities—and up to 23% in deeper dependency layers—remain unpatched due to broken patch flows after base EOL, even when fixes are available upstream. These effects are amplified for vulnerabilities originating far down the base image chain, where fragmented responsibility and unclear maintenance status of tags make remediation less likely. (2) manual remediation is underutilized: explicit updates address only 4.5% of inherited vulnerabilities, usually months late, while overwrites resolve just 3.3%, though immediate overwrites at inheritance can be effective. Maintainers rarely apply these fixes in practice, relying mainly on implicit updates. (3) tag generality is a strong predictor of patching, with highly specific tags having 35% decreased risk to receive timely updates.

To address these, we recommend integrating EOL tracking into CI/CD pipelines and lineage-based base selection, increasing the adoption of overwrite mechanisms, and release our tool and dataset to help the community advance container security.

9. Ethics Considerations

All data analyzed in this study were collected from publicly accessible registries and vulnerability databases, including Docker Hub, the National Vulnerability Database (NVD), and related software repositories. The study involved only known, disclosed CVEs. We also contacted Docker Official Image maintainers prior to publication to share aggregate findings and recommendations regarding EOL handling and tag management, ensuring responsible and constructive community engagement.

10. Acknowledgment

The authors thank the anonymous reviewers for their insightful and constructive feedback, and the shepherd for their guidance in refining the paper throughout the revision process. The authors gratefully acknowledge the support of DARPA (Young Faculty Award), NSF CAREER Award (2427783), and an Amazon Research Award. This work is also supported by the Ministry of Trade, Industry & Energy of the Republic of Korea, via the Industrial Technology Innovation Program (RS-2024-00443436, Development of Integrated Cybersecurity Technology and Evaluation Framework for Bi-directional Charging Systems in Response to Global Regulations). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] D. Lorenc and M. Kaczorowski, "Exploring container security: How containers enable passive patching and a better model for supply chain security | Google Cloud Blog — cloud.google.com," https://cloud.google.com/blog/products/containers-kubernetes/exploring-container-security-how-containers-enable-passive-patching-and-a-better-model-for-supply-chain-security, 2018, [Accessed 21-01-2024].

[2] S. J. Bigelow, "What is immutable infrastructure? | Definition from TechTarget — techtarget.com," https://www.techtarget.com/searchitoperations/definition/immutable-infrastructure, 2022, [Accessed 21-01-2024].

[3] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2187–2200. [Online]. Available: https://doi.org/10.1145/3133956.3134059

[4] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitraş, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *S&P*, 2015.

[5] D. K. Mulligan and F. B. Schneider, "Doctrine for cybersecurity," *Daedalus, Journal of the American Academy of Arts and Sciences*, vol. 140, no. 4, pp. 70–92, 2011.

[6] Federal Register, "Improving the nation's cybersecurity," https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity, 2021, accessed: 2025-04-13.

[7] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269–280.

[8] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the relation between outdated docker containers, severity vulnerabilities, and bugs," in *2019 ieee 26th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 2019, pp. 491–501.

[9] J. Luu, "A deep dive into docker hub's security landscape-a story of inheritance?" Master's thesis, 2019.

[10] K. Wist, M. Helsem, and D. Gligoroski, "Vulnerability analysis of 2500 docker hub images," in *Advances in Security, Networks, and Internet of Things: Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20*. Springer, 2021, pp. 307–327.

[11] A. Zerouali, T. Mens, A. Decan, J. Gonzalez-Barahona, and G. Robles, "A multi-dimensional analysis of technical lag in debian-based docker images," *Empirical Software Engineering*, vol. 26, no. 2, p. 19, 2021.

[12] M. U. Haque and M. A. Babar, "Well begun is half done: An empirical study of exploitability & impact of base-image vulnerabilities," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 1066–1077.

[13] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang, W.-H. Lee, T. Lu, W. Chen, and R. Beyah, "Understanding the security risks of docker hub," in *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*. Springer, 2020, pp. 257–276.

[14] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras, "From patching delays to infection symptoms: using risk profiles for an early discovery of vulnerabilities exploited in the wild," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 903–918.

[15] N. McDonald, S. Schoenebeck, and A. Forte, "Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. CSCW, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3359174

[16] E. Rescorla, "Security holes... who cares," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 75–90.

[17] S. Frei, "Security econometrics - the dynamics of (in)security," ETH Zurich, Dissertation 18197, ETH Zurich, 2009. [Online]. Available: http://www.techzoom.net/publications/security-econometrics/

[18] M. Shahzad, M. Z. Shafiq, and A. X. Liu, “A large scale exploratory analysis of software vulnerability life cycles,” in *Proceedings of the 2012 International Conference on Software Engineering*, 2012.

[19] F. Li and V. Paxson, “A large-scale empirical study of security patches,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[20] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser, “How long do vulnerabilities live in the code? a Large-Scale empirical measurement study on FOSS vulnerability lifetimes,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 359–376. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos

[21] E. Rescorla, “Is finding security holes a good idea?” in *IEEE Security and Privacy*, 2005.

[22] K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitraş, “Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild,” in *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses*, Gothenburg, Sweeden, Sep 2014.

[23] M. Coemans, G. Verbeke, B. Döhler, C. Süsal, and M. Naesens, “Bias by censoring for competing events in survival analysis,” *BMJ*, vol. 378, 2022. [Online]. Available: https://www.bmj.com/content/378/bmj-2022-071349

[24] T. Therneau, C. Crowson, and E. Atkinson, “Multi-state models and competing risks,” *CRAN-R (https://cran. r-project. org/web/packages/survival/vignettes/compete. pdf)*, 2020.

[25] D. Lorenc and M. Kaczorowski. (2018, Dec.) Exploring container security: Let google do the patching with new managed base images. Google Cloud Blog, Containers & Kubernetes. [Online]. Available: https://cloud.google.com/blog/products/containers-kubernetes/exploring-container-security-let-google-do-the-patching-with-new-managed-base-images

[26] d. team, “Why does my security scanner show that an image has cves?” GitHub FAQ: docker-library/faq, 2025, https://github.com/docker-library/faq#why-does-my-security-scanner-show-that-an-image-has-cves. [Online]. Available: https://github.com/docker-library/faq#why-does-my-security-scanner-show-that-an-image-has-cves

[27] “docker-library/python — github.com,” https://github.com/docker-library/python/issues, [Accessed 07-08-2025].

[28] “nodejs/docker-node — github.com,” https://github.com/nodejs/docker-node/issues/, [Accessed 07-08-2025].

[29] “nginx/docker-nginx — github.com,” https://github.com/nginx/docker-nginx/issues/, [Accessed 07-08-2025].

[30] A. McDonald, C. Barwulor, M. L. Mazurek, F. Schaub, and E. M. Redmiles, “"it’s stressful having all these phones": Investigating sex workers’ safety goals, risks, and practices online,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 375–392. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/mcdonald

[31] X. Bouwman, H. Griffioen, J. Egbers, C. Doerr, B. Klievink, and M. van Eeten, “A different cup of TI? the added value of commercial threat intelligence,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 433–450. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/bouwman

[32] “libheif1:1.15.1-1 used includes fixable vulnerabilitites,” https://github.com/nginx/docker-nginx/issues/941, 2024, accessed: 2024-08-07.

[33] Docker Library Maintainers, “Python 3.5 deprecation,” https://github.com/docker-library/python/issues/556, October 2020, gitHub Issue #556, accessed November 11, 2025.

[34] “i386/ubuntu latest tag outdated,” https://github.com/docker-library/official-images/issues/19125, 2025, accessed: 2024-08-07.

[35] “Is the centos image still maintained,” https://github.com/docker-library/official-images/issues/11679, 2022, accessed: 2024-08-07.

[36] “CVE-2021-46828: libtirpc denial-of-service vulnerability,” https://nvd.nist.gov/vuln/detail/CVE-2021-46828, 2021, accessed: 2025-02-10.

[37] D. G. Kleinbaum and M. Klein, *Survival Analysis: A Self-Learning Text*, 3rd ed. Springer, 2011.

[38] P. C. Austin, D. S. Lee, and J. P. Fine, “Introduction to the analysis of survival data in the presence of competing risks,” *Circulation*, vol. 133, no. 6, pp. 601–609, 2016. [Online]. Available: https://www.ahajournals.org/doi/abs/10.1161/CIRCULATIONAHA.115.017719

[39] H. Rossman, A. Keshet, and M. Gorfine, “Pymsm: Python package for competing risks and multi-state models for survival data,” *Journal of Open Source Software*, 2022. [Online]. Available: https://doi.org/10.21105/joss.04566

[40] I. Docker, “Docker Official Images,” https://github.com/docker-library/official-images, 2024, available at https://github.com/docker-library/official-images. [Online]. Available: https://github.com/docker-library/official-images

[41] D. Library, “Docker Official Images: Repo-info,” https://github.com/docker-library/repo-info, 2024, available at https://github.com/docker-library/repo-info. [Online]. Available: https://github.com/docker-library/repo-info

[42] Aqua Security, “Trivy: Vulnerability Scanner for Containers and other Artifacts,” https://github.com/aquasecurity/trivy, 2024, available at https://github.com/aquasecurity/trivy. [Online]. Available: https://github.com/aquasecurity/trivy

[43] A. Security, “Trivy — vulnerability scanner — user guide,” 2025, accessed: 2025-11-13. [Online]. Available: https://trivy.dev/docs/latest/guide/scanner/vulnerability/

[44] A. D. Magalhaes, “Websphere liberty tags not updated,” https://github.com/docker-library/official-images/issues/5850, 2019, accessed: 2024-03-24.

[45] L. Jones, “Lots of arm64v8 images were cancelled - not re-built since,” https://github.com/docker-library/official-images/issues/4881, 2018, accessed: 2024-03-24.

[46] D. Curylo, “Missing arm64v8 job for swipl,” https://github.com/docker-library/official-images/issues/4555, 2018, accessed: 2024-03-24.

[47] Docker PHP Maintainers, “Issue #1178: PHP Docker Image Issue,” https://github.com/docker-library/php/issues/1178, 2024, available at https://github.com/docker-library/php/issues/1178. [Online]. Available: https://github.com/docker-library/php/issues/1178

[48] Node.js Docker Maintainers, “Issue #1525: Node.js Docker Image Issue,” https://github.com/nodejs/docker-node/issues/1525, 2024, available at https://github.com/nodejs/docker-node/issues/1525. [Online]. Available: https://github.com/nodejs/docker-node/issues/1525

[49] pospjan, “Current version of adminer generates warnings under php 8 when submitting custom queries,” 2021, accessed: 2024-11-14. [Online]. Available: https://github.com/TimWolla/docker-adminer/issues/108

[50] National vulnerability database, http://nvd.nist.gov/.

[51] L. Bilge and T. Dumitraş, “Before we knew it: an empirical study of zero-day attacks in the real world,” in *ACM Conference on Computer and Communications Security*, 2012, pp. 833–844.

[52] CISA, “BOD 19-02: Vulnerability remediation requirements for internet-accessible systems,” 2019, accessed: 2024-11-14. [Online]. Available: https://www.cisa.gov/news-events/directives/bod-19-02-vulnerability-remediation-requirements-internet-accessible-systems

[53] ExploitDB, “The exploit database,” https://www.exploit-db.com/, 2019.

[54] National Vulnerability Database, “CVE-2022-23218: Buffer Overflow in GNU C Library’s sunrpc Module,” https://nvd.nist.gov/vuln/detail/CVE-2022-23218, 2022, accessed: 2024-11-13.

[55] Docker Inc., “Docker and snyk announce partnership to provide container vulnerability scanning,” https://www.docker.com/press-release/docker-snyk-announce-partnership-container-vulnerability-scanning/, 2020, accessed: 2024-08-07.

[56] S. Abd ElHafeez, G. D'Arrigo, D. Leonardis, M. Fusaro, G. Tripepi, and S. Roumeliotis, “Methods to analyze time-to-event data: The cox regression analysis,” *Oxidative medicine and cellular longevity*, vol. 2021, no. 1, p. 1302811, 2021.

[57] National Vulnerability Database, “Cve-2024-3094 detail,” https://nvd.nist.gov/vuln/detail/CVE-2024-3094, 2024, accessed: 2026-03-10.

[58] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, “A measurement study on linux container security: Attacks and countermeasures,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[59] A. Y. Wong, E. G. Chekole, M. Ochoa, and J. Zhou, “Threat modeling and security analysis of containers: A survey,” *arXiv preprint arXiv:2111.11475*, 2021.

[60] M. Dahlmanns, C. Sander, R. Decker, K. Wehrle, J. Pennekamp, A. Belova, T. Bergs, M. Bodenbenner, A. Bührig-Polaczek, I. Kunze *et al.*, “Secrets revealed in container images: An internet-wide study on occurrence and impact,” in *ACM Transactions on Internet Technology*, no. 252-266. ACM, 2023, pp. 252–266.

[61] N. Spahn, N. Hanke, T. Holz, C. Kruegel, and G. Vigna, “Container orchestration honeypot: Observing attacks in the wild,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 381–396.

[62] P. Ladisa, H. Plate, M. Martinez, and O. Barais, “Sok: Taxonomy of attacks on open-source software supply chains,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1509–1526.

[63] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, “Towards measuring supply chain attacks on package managers for interpreted languages,” in *Network and Distributed Systems Security Symposium*, 2021.

[64] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, “An empirical analysis of the docker container ecosystem on github,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 323–333.

[65] B. Tak, H. Kim, S. Suneja, C. Isci, and P. Kudva, “Security analysis of container images using cloud analytics framework,” in *Web Services–ICWS 2018: 25th International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings 16*. Springer, 2018, pp. 116–133.

[66] C. Lin, S. Nadi, and H. Khazaei, “A large-scale data set and an empirical study of docker images hosted on docker hub,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 371–381.

[67] A. Sarabi, Z. Zhu, C. Xiao, M. Liu, and T. Dumitraş, “Patch me if you can: A study on the effects of individual user behavior on the end-host vulnerability state,” in *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings 18*. Springer, 2017, pp. 113–125.

[68] E. M. Redmiles, Z. Zhu, S. Kross, D. Kuchhal, T. Dumitras, and M. L. Mazurek, “Asking for a friend: Evaluating response biases in security user studies,” in *Proceedings of the 2018 acm sigsac conference on computer and communications security*, 2018, pp. 1238–1255.

[69] D. Arya, W. Wang, J. L. Guo, and J. Cheng, “Analysis and detection of information types of open source software issue discussions,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 454–464.

[70] J.-U. Holtgrave, K. Friedrich, F. Fischer, N. Huaman, N. Busch, J. H. Klemmer, M. Fourné, O. Wiese, D. Wermke, and S. Fahl, “Attributing open-source contributions is critical but difficult: A systematic analysis of github practices and their impact on software supply chain security.” in *NDSS*, 2025.

[71] Docker Library Contributors, “PR #2205: Update to Docker Library Documentation,” https://github.com/docker-library/docs/pull/2205, 2022, accessed: 2024-11-13.

[72] Docker Official Images Contributors, “Commit 305a689: Modifications to Docker Official Images Documentation,” https://github.com/docker-library/official-images/commit/305a689a9b8ced6df9a4fa1dbcc6ecb34aa6c9ee#diff-7077506a2e33dccfe04067c214e487097a88497efd08c9ffcfb573f04e1bd86cL50, 2023, accessed: 2024-11-13.

[73] National Vulnerability Database, “CVE-2021-3918: Vulnerability in json-schema Library,” https://nvd.nist.gov/vuln/detail/CVE-2021-3918, 2021, accessed: 2024-11-13.

[74] JSON Schema Contributors, “JSON Schema Specification,” https://github.com/json-schema-org/json-schema-spec, accessed: 2024-11-13.

Appendix A. Additional Details

A.1. CentOS Case

The (`swift:centos7-slim`, `linux`, `amd64`) lineage inherited the high-severity CVE-2022-23218 [54] from its base image `centos:7` on its first image pushed on Aug 11 2020. On Sep 15, 2021 the base lineage reached EOL, while still including the vulnerability, as the project was deprecated. However, the deprecation notice for the CentOS the project (and its tags) was added much later, on Sep 29, 2022 [71]. The vulnerability was disclosed on Jan 14, 2022. On Mar 31, 2023, the tag reached EOL [72], after containing the critical vulnerability for 441 days post-disclosure, inheriting it from an outdated base image. This case underscores the importance of proactive notices for users of the tags, as well as the necessity of frequent scans for supply chain vulnerabilities.

```
Maintainers: Jane Doe <janedoe@myapp.com> (@janedoe)

Tags: 1.0.0, 1.0, 1
Architectures: amd64, arm64v8
GitRepo: https://github.com/myapp/myapp.git
GitCommit: 5b0b75ffdc67ec241739
amd64-Directory: amd64/
arm64v8-Directory: armv8/

Tags: 2.1.0, 2.1, 2
Architectures: amd64, arm64v8
GitRepo: https://github.com/myapp/myapp.git
amd64-GitCommit: 5b0b75ffdc67ec24173
amd64-GitCommit: 10ac8933ffec2467162
amd64-Directory: amd64/
arm64v8-Directory: armv8/
```

Listing 1: CI/CD metadata file example

A.2. CVE-2021-3918 in Node.js.

We illustrate our analysis of lifespan of owned vulnerabilities leveraging CVE-2021-3918 [73], a vulnerability in the `json-schema` [74] library prior to version 0.4.0, in Node.js[6] images (see Figure 7 for the timeline). The

6. It is a popular server runtime environment having over 5 billion pulls on Docker Hub, used commonly as a base image for web applications.

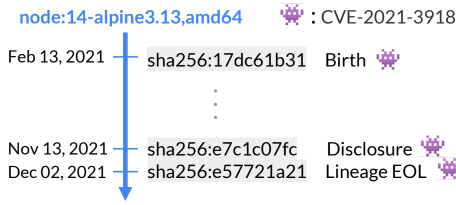


Figure 7: The lifespan of CVE-2021-3918 in json-schema in amd64 Node.js images with tag `14-alpine3.13`.

subject of this analysis is CVE-2021-3918 in the lineage of Node.js application with tag `14-alpine.13` for `amd64`. The vulnerability was present in the lineage since its inception, on Feb 13, 2021. It was not publicly known at that time (or a zero-day vulnerability) [51]. Later, on Nov 1, 2021, the vulnerability was patched with version 0.4.0 of `json-schema` package. Developers using this lineage in their own containers were then faced with two possible outcomes. If the Node.js maintainers would eventually patch CVE-2021-3918 in this lineage, then the patch would be received automatically. Instead, the lineage reached EOL 17 days later, with the vulnerability still being present. In this case, patching the vulnerability requires an explicit base update for all the images that are `FROM` the EOL lineage. This illustrates the key insight of our model: because the "Death" and "Lineage EOL" states represent *competing risks*, where each state precludes the other one from occurring, the end-users of Docker images should expect vulnerabilities to remain unpatched unless the users take an explicit action.

Table 3: Hazard Ratios for Implicit Fixes (N=1,000,000)

Covariates	Units / Comparison	HR
Base maintainer is Docker community	Docker vs upstream	0.44
Repo maintainer is Docker community	Docker vs upstream	0.97
Number of super lineages of the base	1 more super	0.94
Repo pull count: Moderate	vs Low	1.13
Repo pull count: High	vs Low	0.90
Base repo pull count: Moderate	vs Low	1.18
Base repo pull count: High	vs Low	1.23
Arch = s390x	vs amd64	1.20
Arch = 386	vs amd64	1.17
Arch = ppc64le	vs amd64	1.15
Arch = arm	vs amd64	1.13
Arch = mips64le	vs amd64	1.09
Base severity = HIGH	vs MEDIUM	1.03
Base severity = CRITICAL	vs MEDIUM	1.25
Base severity = LOW	vs MEDIUM	0.58

A.3. Qualitative Analysis Detail for Patching Delay Cause

Table 4 shows the distribution of issues across the analyzed repositories. Table 5 summarizes the codebook generated by two independent researchers to analyze Github issues and identify the root cause of patching delays. Because any given issue can be stalled for several reasons, multiple codes can be assigned. For example, an issue open in the wrong repository and still waiting for its base image to be rebuilt can carry both codes (waiting on a base image update and confusion about tag's maintenance status). After conducting separate coding rounds, the researchers reconciled and agreed on eight distinct codes.

Table 4: Distribution of Issues by Repository

Repository	Issues
Node	9
Python	15
Nginx	15
official-images	18
Total	54

Table 5: Summary of Codes Identified Patching Delay Cause

Code	# Counts
Waiting on a base image update	27
Confusion about who is responsible	9
Not fixed in the upstream package	7
Confusion about tag's maintenance status	6
Infrastructure or CI/CD problem	6
Won't be fixed in the upstream package	3
Base image update needed	2
Breaking Change	1

- **Waiting on a base image update**: The maintainer prefers to wait until the base image is rebuilt with patched packages.
- **Not fixed in the upstream package**: The maintainer cannot address the issue since there is no available fixed version of the upstream package.
- **Won't be fixed in the upstream package**: The issue is not slated for a fix in the upstream package, so maintainers cannot address it.
- **Base image update needed**: The maintainer needs to update the base image manually (i.e., changing the tag).
- **Confusion about tag's maintenance status**: The user expresses a confusion over why the tag is outdated.
- **Confusion about who is responsible**: The responsibility to patch the vulnerability is unclear to the user. Issue is filed in the wrong repository or assigned to maintainers who will not act, necessitating redirection to the appropriate parties.
- **Infrastructure or CI/CD problem**: There is a delay due to the build issues, such as the build servers being down or CI/CD build failures.
- **Breaking change**: The patch introduces breaking changes, and maintainers delay the adoption to avoid incompatibility or disruptions to users.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper presents an empirical study of container images to investigate whether, and how promptly, vulnerabilities are patched through different update mechanisms, including automatic (implicit) updates, explicit (manual) updates, and independent overwrites. To this end, the authors collect a large set of container images and measure patching delays as well as the impact of delayed remediation. The main findings are as follows: (1) the end-of-life status of base images poses a major obstacle to automated updates; (2) manual remediation via explicit updates is both infrequent and significantly delayed; and (3) patch propagation is strongly influenced by the specificity of image tags.

B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research.
- Provides a New Data Set For Public Use.
- Creates a New Tool to Enable Future Science.
- Addresses a Long-Known Issue.
- Provides a Valuable Step Forward in an Established Field.
- Establishes a New Research Direction.

B.3. Reasons for Acceptance

1) This work provides important and novel insights, showing that the tag-based versioning system used in Docker images is fundamentally flawed and may inherently delay patch deployment.
2) The paper suggests introducing lineage-based information as complementary metadata to eliminate ambiguity in version control.

B.4. Noteworthy Concerns

1) The evaluation could be broader and more in-depth. Expanding coverage across more container image categories would better justify the representativeness of the study subjects. Additionally, a more granular analysis of how delayed patching affects different image categories could provide deeper insights into the varying levels of maintenance support across these classes.
2) The study could also benefit from being conducted on more recent datasets to better reflect current practices in container image maintenance.