Supporting Upstream Vulnerability Triage in Fork-Based Development: A Brazilian Government Experience

José Renan F. Damasceno[1], **Lincoln Rocha**[1,2], **Paulo Rego**[1,2]

[1]Postgraduate Program in Computer Science – Federal University of Ceará (UFC)
Zip Code 60.440-900 – Fortaleza – CE – Brazil

[2]Department of Computing – Federal University of Ceará (UFC)
Zip Code 60.440-900 – Fortaleza – CE – Brazil

joserenandamasceno@alu.ufc.br, {lincoln,paulo}@dc.ufc.br

Abstract. *Open-source software reuse accelerates the development of complex systems, but it also introduces software supply chain risks when downstream projects evolve independently from their upstream codebases. This challenge is particularly relevant in fork-based development, where security fixes disclosed upstream must be assessed, prioritized, and adapted under project-specific constraints. This paper reports a Brazilian government experience in monitoring upstream vulnerabilities for a secure mobile communication app derived from Matrix/Element open-source clients. We present an automated workflow that combines CPE-based queries to the National Vulnerability Database with advisory mining from the GitHub Advisory Database, enriches vulnerability records with CWE information, and extracts metadata from upstream vulnerability-fixing commits. Rather than replacing human assessment, the workflow provides structured evidence for AppSec-mediated decision-making among development, security testing, and DevSecOps teams. The experience shows how upstream vulnerability monitoring can support downstream triage, remediation planning, patch inspection, and documented downstream security assessment. Our findings highlight that vulnerability management in fork-based projects is not only a technical detection problem, but also an organizational coordination challenge involving evidence-driven, contextual analysis, and security governance.*

1. Introduction

Open-source software (OSS) reuse has become a fundamental strategy for accelerating software development, reducing implementation costs, and enabling organizations to build on mature and community-tested codebases. This practice is particularly relevant in security-sensitive domains, where adopting established projects can provide access to complex features, active maintenance, and previous security hardening efforts. However, OSS reuse also introduces software supply chain risks: vulnerabilities disclosed in reused components may propagate to downstream systems, especially when the reused code is modified, embedded, or evolved independently from its original project [Woo et al. 2021, Williams et al. 2025, Alomari et al. 2025].

Fork-based development is a common form of OSS reuse in which a downstream project starts from an existing repository and evolves according to its own technical, organizational, or regulatory requirements. Although this model supports customization and autonomy, it also creates long-term maintenance challenges. Divergent forks

accumulate architectural and implementation differences over time, making upstream integration increasingly costly and conflict-prone [Salman 2021, Businge et al. 2022, Sung et al. 2020]. This challenge is even more critical when upstream changes involve security fixes. Downstream maintainers must assess whether a disclosed vulnerability affects their baseline, understand its severity and weakness category, identify the relevant fix, and estimate the effort needed to integrate or adapt it.

Existing vulnerability databases and advisory sources, such as the National Vulnerability Database (NVD) and the GitHub Advisory Database, provide valuable information about publicly disclosed vulnerabilities, including identifiers, severity scores, affected products, weakness categories, and external references. Prior work has investigated vulnerable version identification, vulnerability-fixing commits, and large-scale datasets linking CVEs to source-code changes [Sun et al. 2022, Cheng et al. 2026, Li and Paxson 2017, Bhandari et al. 2021, Akhoundali et al. 2024, Hommersom et al. 2024]. Existing approaches do not primarily support downstream maintainers of divergent forks in deciding whether, where, and how to integrate upstream security fixes. In such settings, vulnerability monitoring must go beyond detecting vulnerable dependencies: it must correlate advisory records, affected upstream versions, weakness categories, and fix-commit evidence under the constraints of a modified codebase.

This paper reports an experience from a Brazilian government secure communication project developed as a downstream fork of open-source Matrix ecosystem clients. Matrix is a decentralized communication protocol designed to support federated, interoperable, and secure real-time communication [Martins et al. 2026]. In our context, the adopted OSS baseline enabled the construction of a mobile government application, but also made upstream vulnerability monitoring a recurring security activity. Since the downstream application evolved under institutional, security, and integration requirements, simply tracking upstream releases was insufficient. The team needed a systematic way to identify relevant upstream vulnerabilities, classify their security implications, and inspect the available fixing evidence before planning remediation.

To address this need, we present an automated vulnerability monitoring workflow for fork-based development. The workflow combines CPE-based queries to the NVD with advisory mining from the GitHub Advisory Database, enriches the collected vulnerabilities with CWE information, and extracts metadata from vulnerability-fixing commits available in upstream repositories. In our experience, this workflow enabled the team to consolidate vulnerability information from multiple sources, identify cases that required advisory-based enrichment beyond CPE-based NVD queries, and estimate the scope of available fixes through commit-level metadata.

The main contributions of this paper are: (i) a vulnerability monitoring workflow for downstream fork-based projects that combines NVD, CPE, GitHub Advisory Database, CWE, and upstream fix-commit metadata; (ii) a Brazilian government case study applying this workflow to a secure mobile communication app derived from Matrix/Element clients; (iii) evidence that AppSec-mediated analysis can support coordinated triage, remediation planning, patch adaptation, and risk acceptance; and (iv) lessons for teams maintaining forked OSS codebases in security-sensitive environments.

The remainder of this paper is organized as follows. Section 2 presents back-

ground on vulnerability databases and fork-based development. Section 3 characterizes the government project and its OSS baseline. Section 4 describes the proposed vulnerability monitoring workflow. Section 5 presents the results, lessons learned, and implications. Section 6 discusses related work, Section 7 presents threats to validity and Section 8 concludes the paper and outlines future work.

2. Background

2.1. Vulnerability Databases

The Common Vulnerabilities and Exposures[1] (CVE) program assigns standardized identifiers to publicly disclosed cybersecurity vulnerabilities. The National Vulnerability Database[2] (NVD), maintained by the U.S. National Institute of Standards and Technology (NIST), enriches CVE records with metadata such as Common Vulnerability Scoring System (CVSS) metrics, external references, associated weaknesses, and affected products. CVSS provides numerical and qualitative severity ratings, whereas the Common Platform Enumeration (CPE) provides a structured naming scheme for identifying affected products and versions. The NVD makes these records available through a REST API.

The Common Weakness Enumeration[3] (CWE) classifies recurring software and hardware weaknesses that may contribute to vulnerabilities. While a CVE represents a specific publicly disclosed vulnerability, a CWE describes the underlying weakness category. A CVE may therefore be associated with one or more CWE identifiers, allowing vulnerabilities to be analyzed according to broader security concerns.

The GitHub Advisory Database[4] provides security advisories for open-source projects, including CVE-associated and GitHub-originated records identified by GitHub Security Advisory (GHSA) identifiers. These advisories can complement NVD searches because some vulnerabilities associated with upstream repositories are not properly linked to product CPEs. Our workflow therefore combines NVD/CPE queries with GitHub Advisory Database mining to improve the collection of relevant upstream vulnerability evidence.

2.2. Fork-Based Development

Despite its evident benefits such as accelerating the development of new applications [Woo et al. 2021, Williams et al. 2025], the reuse of unmanaged Open Source Software (OSS) components introduces critical threats to the software supply chain, particularly regarding the propagation of vulnerabilities [Alomari et al. 2025]. The literature indicates that reuse rarely occurs through the simple importation of unchanged libraries; instead, third-party code is predominantly modified and adapted to new contexts. This common practice of cloning and modifying code obscures the original software provenance, making it difficult to trace inherited security flaws [Woo et al. 2021, Sun et al. 2022].

Social coding platforms have consolidated forking as a mechanism for enabling and managing large-scale open-source code reuse. Forking creates a complete copy of a

[1] `https://www.cve.org/`
[2] `https://nvd.nist.gov/`
[3] `https://cwe.mitre.org/`
[4] `https://github.com/advisories`

repository, including its source code and version history, allowing developers to pursue an independent line of work without affecting the stability of the original project, known as the upstream [Salman 2021]. The literature classifies forks according to their intended relationship with the upstream. Social forks are temporary repositories created to implement features or fix defects, with the intention of contributing changes back through Pull Requests [Imamura et al. 2022]. Hard forks, in contrast, are created to establish autonomous projects without an expectation of reintegration, often due to technical disagreements or organizational requirements [Businge et al. 2022]. Within this category, divergent forks represent a critical subcategory: they evolve independently over time, accumulate substantial architectural changes, may shift from the original purpose, and make synchronization with the upstream increasingly complex and conflict-prone [Businge et al. 2022].

Maintaining derivative repositories requires the continuous propagation of changes. In ecosystems characterized by divergent forks, this process often follows an asymmetric code flow: downstream projects frequently need to incorporate security updates and other changes from the upstream project, whereas the reverse flow rarely occurs [Salman 2021]. In this context, code synchronization often moves beyond traditional Pull Request mechanisms. According to [Businge et al. 2022], fork maintainers typically rely on version-control operations such as merge, rebase, and cherry-pick to integrate upstream updates. However, high structural divergence makes these operations technically costly. Integration attempts may cause conflicts, build or test failures, and often require extensive manual refactoring [Sung et al. 2020, Imamura et al. 2022].

3. Project Characterization

In this section, we present an overview of the MSG project, including its institutional context, technical foundations (underlying technologies and the base projects), and organizational structure (the mobile app security squad involved in the initiative).

3.1. Overview

This work is conducted within the scope of an institutional collaboration between Federal University of Ceará (UFC), the Brazilian Intelligence Agency (ABIN), and the Federal Data Processing Service (SERPRO), under the MSG project. The project began in March 2024 and was completed in July 2026. This partnership integrates academic research, governmental expertise, and technological infrastructure to develop solutions for secure communication and strategic information management in mobile government (m-gov).

UFC is responsible for the scientific and methodological development, including computational models, data analysis techniques, and experimental validation. ABIN defines operational requirements and provides the security and intelligence context. SERPRO provides the technological infrastructure, enabling scalable system development and integration with federal government environments. This collaboration fosters knowledge transfer between academia and the public sector, strengthening the country's sovereignty.

The MSG app is a secure communication platform designed for the Brazilian federal public service. It is aligned with Gov.br standards and supports messaging, media sharing, audio and video calls, location sharing, and group communication. Although the app is publicly available for download, its use is restricted to authorized government

personnel. It is available on the Android (Google Play[5]) and iOS (Apple Store[6]) stores.

3.2. The Base Projects

The MSG app was developed based on established open-source solutions within the Matrix ecosystem[7], particularly Element's clients for Android[8] and iOS[9]. The Matrix protocol has emerged as a decentralized alternative for real-time communication [Martins et al. 2026] and is widely adopted in scenarios that require digital sovereignty, interoperability, and security, especially in governmental and institutional initiatives across several countries in Europe (e.g., Germany, France, Sweden, Netherlands, and Switzerland) [Element 2023].

Matrix is an open protocol that enables federated communication [Martins et al. 2026], eliminating reliance on centralized service providers and allowing organizations to retain control over their own data and infrastructure. This model has proven particularly relevant in the European context, where there is increasing concern regarding digital sovereignty, data protection, and regulatory compliance (GDPR). Thus, there is a growing adoption of open, auditable, and resilient technologies.

The Element's clients, which are reference implementations of the Matrix ecosystem, provide native support for end-to-end encryption (E2EE), secure communication, and integration with federated servers. We developed the MSG app versions baseline from the Element Android (v1.6.20) and iOS (v1.11.15) codebases. This version was selected based on technical criteria, including stability, code maturity, compatibility with essential Matrix protocol features, and licensing model.

At the time of adoption, the Element Android and iOS repositories had approximately 3.7k and 1.8k stars on GitHub, respectively, indicating a certain level of community adoption and trust. In addition, both projects present an extensive commit history and regular release cycles, reflecting active maintenance and continuous evolution. Publicly disclosed vulnerabilities, identified as CVEs, were also considered, enabling an assessment of the projects' security history and the maintainers' responsiveness to incidents.

In addition to the Matrix/Element client repositories, the monitoring scope included supporting upstream components that condition the security posture of the downstream application, such as Matrix SDKs, cryptographic libraries, and Keycloak as the identity and access management component used in authentication flows.

3.3. The Mobile App Security Squad

One squad was assigned to ensure the MSG app's compliance with the OWASP MAS standard. The team includes researchers from UFC and industry professionals, working closely with ABIN and SERPRO to align security practices and ensure consistent application of security controls throughout the software development lifecycle.

[5]https://play.google.com/store/apps/details?id=br.gov.comunicacaosegura
[6]https://apps.apple.com/br/app/msg-gov/id6624295607
[7]https://matrix.org/ecosystem/
[8]https://github.com/element-hq/element-android
[9]https://github.com/element-hq/element-ios

The squad is structured into distinct action fronts, each with specific responsibilities, enabling a comprehensive and specialized approach to mobile app security: **Academic Research**: responsible for investigating new security approaches, tools, and methodologies, as well as adapting state-of-the-art practices to the project context. This line also contributes to the continuous evolution of adopted strategies based on empirical evidence and experimentation; **Application Security (AppSec)**: focuses on defining and implementing security requirements, including activities such as threat modeling, code review, and architectural analysis. Within this line, a designated professional acts as a liaison between the mobile development and security teams, facilitating communication and ensuring the effective integration of security practices into the development process; **Security Testing / Penetration Testing**: responsible for conducting both manual and automated security tests, following established guidelines such as the OWASP MAS. This line emphasizes the identification, validation, and exploitation of vulnerabilities, contributing to a practical assessment of the application's security posture; and **DevSecOps**: integrates security practices into the CI/CD pipeline, including automated analysis (SAST, DAST, and SCA) and security gates to prevent releases with critical vulnerabilities.

4. The Vulnerability Monitoring Workflow

4.1. Overview

Our approach supports downstream developers in identifying, tracking, and analyzing vulnerabilities disclosed in upstream projects. It correlates vulnerability data from the NVD and GitHub Advisory Database with CWE and fixing-commit evidence to support contextualized downstream triage. We investigate the following research questions:

RQ1. What are the characteristics of CVEs affecting the upstream projects? We characterize their sources, severity, criticality, and exploitability indicators.

RQ2. What are the categories of CWEs affecting the upstream projects? We analyze the weakness categories associated with the identified vulnerabilities to reveal recurring security concerns.

RQ3. What are the characteristics of fixes for CVEs affecting upstream projects? We analyze changed files and lines of code as initial indicators of the scope of the available upstream fixes.

4.2. Workflow

The workflow comprises six automated steps, as illustrated in Figure 1. It starts from a `target` JSON file describing the upstream projects to be monitored. Each entry specifies the GitHub repository owner (`github_owner`) and name (`github_repo`), the CPE vendor (`cpe_vendor`) and product (`cpe_product`), and, optionally, the target operating system (`cpe_target_sw`). It also lists the product versions to be monitored (`cpe_versions`). These fields are used to construct a CPE Well-Formed Name (WFN) for each version and query the NVD for associated CVEs.

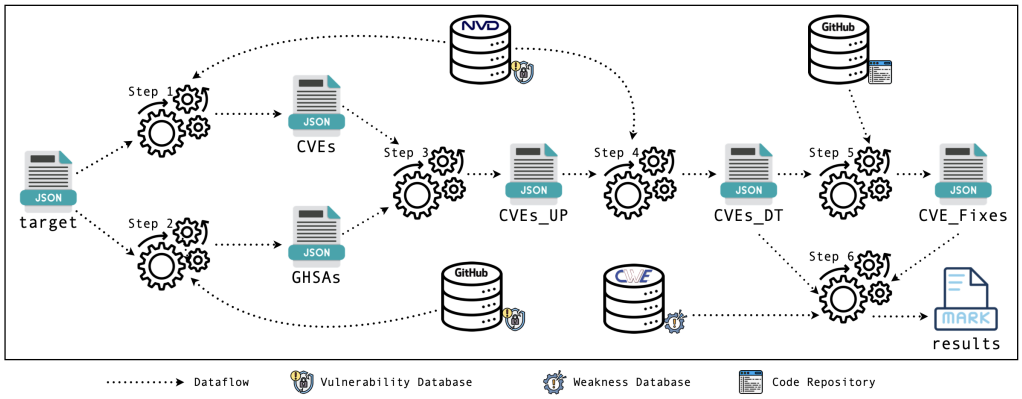

Figure 1. Vulnerability monitoring workflow.

Listing 1 presents an example entry for the Element Android upstream project.

Listing 1. Example entry from the target JSON file.

```
{"github_owner": "element-hq",
 "github_repo": "element-android",
 "cpe_vendor": "element",
 "cpe_product": "element",
 "cpe_target_sw": "android",
 "cpe_versions": ["1.6.58", "1.6.56", "1.6.54", "1.6.52",
                  "1.6.50", "1.6.48", "1.6.46", "1.6.44",
                  "1.6.42", "1.6.40", "1.6.38", "1.6.36",
                  "1.6.34", "1.6.32", "1.6.28", "1.6.26",
                  "1.6.24", "1.6.22", "1.6.20"]}
```

In this entry, `element-hq` and `element-android` identify the GitHub repository, while `element` identifies both the vendor and product in the NVD. The `cpe_target_sw` field restricts the query to Android, and `cpe_versions` defines the monitored versions.

Step 1 reads the `target` file, constructs a CPE WFN for each product version, queries the NVD, and stores the retrieved vulnerabilities in `CVEs`. Step 2 mines the GitHub Advisory Database for advisories associated with each monitored repository and stores them in `GHSAs`. Because some GitHub advisories are not associated with the corresponding product CPE in the NVD, Step 3 retrieves these additional CVEs and merges them with the initial dataset, producing `CVEs_UP`.

Step 4 enriches each record in `CVEs_UP` with its CVSS base score and vector, associated CWEs, external references, and available URLs to upstream vulnerability-fixing commits, producing `CVEs_DT`. Step 5 retrieves commit-level metadata, including the commit URL, changed files, additions, deletions, and total changes, and stores it in `CVE_Fixes`. Finally, Step 6 processes `CVEs_DT` and `CVE_Fixes`, combines them with CWE information, and generates the results used to answer the three RQs described in Section 4.1.

The workflow is implemented as a fully automated set of Python scripts. The `pipeline.py` script creates the intermediate JSON artifacts, while `results.py` processes `CVEs_DT` and `CVE_Fixes`. The `results.ipynb` notebook invokes these functions and presents the results in Markdown format. All scripts, configuration files, col-

lected datasets, and analysis notebooks are publicly available in the replication package.[10]

5. Results, Lessons Learned, and Implications

This section presents the results obtained by applying the vulnerability monitoring workflow to the upstream projects adopted as baselines or supporting components of the Android and iOS variants of the MSG mobile app, lessons learned, and implications.

5.1. Results

5.1.1. RQ1: Characteristics of Upstream Vulnerabilities

The mining process identified 51 CVEs affecting the monitored upstream projects. The severity distribution was concentrated in medium-severity vulnerabilities: 31 CVEs (60.8%) were classified as Medium, 10 (19.6%) as High, 8 (15.7%) as Low, and 2 (3.9%) as Critical. CVSS scores ranged from 1.3 to 9.8, with a mean of 5.66, median of 5.40, and standard deviation of 1.87. Table 1 summarizes this distribution.

Table 1. Severity distribution of upstream CVEs.

Severity	CVEs	Percentage
Low	8	15.7%
Medium	31	60.8%
High	10	19.6%
Critical	2	3.9%
Unknown	0	0.0%
Total	51	100.0%

Table 2. CVEs identified per monitored upstream project.

Upstream project	CVEs	Percentage
keycloak	29	56.9%
matrix-rust-sdk	8	15.7%
olm	5	9.8%
element-android	4	7.8%
matrix-ios-sdk	2	3.9%
vodozemac	2	3.9%
element-ios	1	2.0%
Total	51	100.0%

The distribution (see Table 2) by upstream project was uneven. Keycloak accounted for 29 CVEs (56.9%), followed by matrix-rust-sdk with 8 CVEs (15.7%), olm with 5 CVEs (9.8%), element-android with 4 CVEs (7.8%), matrix-ios-sdk and vodozemac with 2 CVEs each (3.9%), and element-ios with 1 CVE (2.0%).

As an illustrative case, the Element Android upstream project shows why the workflow combines CPE-based NVD mining with GitHub Advisory Database mining. Among the four CVEs identified for this upstream project, CVE-2025-27606 was retrieved from both the NVD/CPE query and GitHub Advisory GHSA-632v-9pm3-m8ch. In contrast, CVE-2024-26132, CVE-2024-26131, and CVE-2021-40824 entered the dataset through GitHub advisories, showing that relying only on CPE mappings would have missed relevant mobile-client vulnerabilities. The collected evidence also supported downstream triage by combining severity, weakness, and fix metadata: CVE-2025-27606 was classified as Medium severity, associated with CWE-488, and had two fixing commits; CVE-2024-26132 was classified as Medium severity, associated with CWE-200, and had one fixing commit; CVE-2024-26131 was classified as High severity, associated with CWE-940 and CWE-923, and had one fixing commit; and CVE-2021-40824

[10] `https://github.com/lincolnrocha/PaperSBSeg2026`

was classified as Medium severity, associated with CWE-290, but had no fixing commit identified by the workflow. This case illustrates how the workflow consolidates heterogeneous vulnerability evidence before AppSec triage, instead of treating NVD records, GitHub advisories, CWE categories, and fixing commits as disconnected artifacts.

From the AppSec perspective, this consolidated evidence reduced the need to manually inspect disconnected sources and allowed the team to assess a structured record containing source, severity, CWE classification, and fixing evidence. These results show that the security exposure of a downstream mobile application is not restricted to its client repositories. Supporting components involved in authentication, cryptography, and protocol integration also contributed to the monitoring scope. This is particularly relevant in fork-based development, where upstream vulnerabilities may affect both the forked mobile clients and related components that shape the downstream system's security posture.

From a maintenance perspective, two observations are important. First, most CVEs were not Critical or High, but Medium-severity issues still required triage because their applicability depends on the downstream architecture, enabled features, authentication flows, deployment assumptions, and local modifications. Second, the presence of High and Critical CVEs reinforces the need for continuous monitoring rather than occasional manual inspection of upstream projects.

5.1.2. RQ2: Weakness Categories Affecting the Upstream Projects

The workflow identified 44 distinct CWE categories associated with the collected CVEs. Among them, 8 appeared in the CWE Top 25 list for 2025 and 10 appeared in the corresponding list for 2024. In addition, 3 CWEs were also present in the CWE Top 10 KEV list for 2025 and 3 in the corresponding list for 2024. The weaknesses appearing in both Top 25 and Top 10 KEV were CWE-787, CWE-79, and CWE-502 in 2025, and CWE-89, CWE-787, and CWE-502 in 2024.[11] However, the most recurrent weaknesses in our dataset were not necessarily the globally highest-ranked ones. This reinforces the need to interpret CWE distributions according to the downstream application context rather than relying only on global weakness rankings.

Table 3 presents the CWE categories associated with at least two CVEs in the dataset. The most frequent weakness was CWE-287, Improper Authentication, associated with six CVEs across matrix-ios-sdk, matrix-rust-sdk, and Keycloak. It was followed by CWE-322, Key Exchange without Entity Authentication, associated with three CVEs. Other recurring categories included CWE-290, CWE-200, CWE-208, and CWE-526, each associated with two CVEs.

The CWE-based view helped the team move beyond isolated CVE inspection. Instead of treating each vulnerability as an independent item, the AppSec analysis could group weaknesses by security concern. Authentication-related weaknesses affected both the Matrix ecosystem and Keycloak, making them relevant to login, session, and identity flows. Cryptographic and timing-related weaknesses affected components such as olm and vodozemac, which are relevant to secure messaging. Information exposure and logging weaknesses also required contextual review because their actual impact depends on

[11] We consulted the MITRE CWE Top 25 and CWE Top 10 KEV lists for 2024 and 2025.

Table 3. Most recurrent CWE categories in the monitored upstream projects.

CWE	Weakness	CVEs
CWE-287	Improper Authentication	6
CWE-322	Key Exchange without Entity Authentication	3
CWE-290	Authentication Bypass by Spoofing	2
CWE-200	Exposure of Sensitive Information	2
CWE-208	Observable Timing Discrepancy	2
CWE-526	Cleartext Storage in Environment Variable	2

what data is processed, stored, or exposed by the downstream application.

5.1.3. RQ3: Characteristics of Upstream Fixes

The workflow mined fixing-commit metadata for 16 of the 51 identified CVEs, represented by 20 fixing-commit entries. The fixes for these CVEs were distributed in matrix-rust-sdk (6), element-android (3), Keycloak (3), matrix-ios-sdk (2), and vodozemac (2). The number of changed files per fixing commit ranged from 1 to 44, with a mean of 6.70, median of 2.00, and standard deviation of 12.95. The number of total changes ranged from 2 to 2,595, with a mean of 313.00, median of 48.50, and standard deviation of 782.27.

Table 4. Summary of vulnerability-fixing commit characteristics.

Metric	Min.	Median	Mean	Max.
Changed files	1	2.00	6.70	44
Line additions	1	32.00	286.15	2396
Line deletions	0	3.50	26.85	199
Total changes	2	48.50	313.00	2595

The difference between the mean and median indicates a skewed distribution: most fixes were small or moderate, but a few large commits substantially increased the average. For example, the same matrix-ios-sdk fixing commit was associated with two CVEs and changed 44 files with 2,595 total changes. In contrast, several fixes in element-android, matrix-rust-sdk, and vodozemac involved one to four files and fewer than 120 total changes. This variation is important for downstream maintainers because fix size is only an initial proxy for integration effort. A small fix may be easy to inspect but still affect security-critical logic, while a large fix may require more extensive integration planning, regression testing, and AppSec review. Thus, the workflow used fixing-commit metadata as a triage signal rather than as an automatic measure of remediation effort.

In addition to vulnerability and fixing-commit metadata, the workflow also supported internal AppSec triage by assigning analysis states to the collected upstream CVEs. At the time of the study, 27 CVEs were deferred for later review or continuous monitoring, 14 required manual modification or contextual adjustment of the automated record, and 10 had already been analyzed by the AppSec team. These labels represent intermediate analysis states rather than final remediation outcomes. Therefore, we do not interpret them as evidence that a CVE was patched, not applicable, risk accepted, or fully resolved. A finer-grained classification of downstream remediation outcomes is left as future work.

5.2. Lessons Learned

Lesson 1: Vulnerability monitoring in forks requires evidence consolidation, not only CVE discovery. The experience showed that downstream teams need more than a list of vulnerabilities. They need a consolidated view that connects CVE identifiers, severity, affected upstream projects, CWE categories, advisories, and fixing commits. Without this consolidation, triage tends to be fragmented across multiple sources and depends heavily on manual cross-checking.

Lesson 2: Upstream vulnerability relevance depends on downstream context. The existence of a CVE in an upstream project does not automatically mean that the downstream application is exploitable or that the upstream patch can be applied directly. Applicability depends on the adopted baseline, enabled features, local modifications, authentication flows, deployment configuration, and whether the vulnerable code path is present in the fork. This reinforces the need for AppSec-mediated interpretation of the collected evidence.

Lesson 3: CWE analysis helps guide security review beyond individual CVEs. The CWE distribution revealed recurring security concerns related to authentication, key exchange, information exposure, timing behavior, and sensitive configuration handling. This view helped the team identify broader areas for review and testing, rather than treating each CVE as an isolated remediation item.

Lesson 4: Fix-commit metadata is useful for prioritization, but insufficient for automatic decisions. Changed files, additions, deletions, and total changes helped estimate the apparent scope of upstream fixes. However, these metrics do not capture semantic complexity, downstream divergence, or security impact. Therefore, fix size should support triage and planning, but not replace manual security assessment.

Lesson 5: AppSec acts as the bridge between automated evidence and coordinated action. The workflow became valuable because its outputs were interpreted by an AppSec role capable of connecting development, security testing, and DevSecOps. This role translated upstream evidence into concrete actions, such as opening remediation tasks, planning patch adaptation, requesting validation tests, monitoring deferred issues, or documenting risk acceptance.

5.3. Implications

For downstream maintainers, upstream vulnerability monitoring should be treated as a continuous maintenance activity. Fork-based products cannot rely only on release notes or generic dependency scanners, because vulnerability applicability depends on the relationship between the upstream baseline and the modified downstream codebase.

For AppSec teams, the main value of automation is not replacing human judgment, but reducing the effort required to collect and correlate evidence. In security-sensitive fork-based projects, AppSec needs enough context to decide whether a vulnerability is applicable, how urgent it is, whether the fix can be applied, and what validation is required. **For DevSecOps pipelines**, the workflow can complement SAST, DAST, SCA, and security gates by adding upstream-specific vulnerability intelligence. This is especially important when reused OSS is not only consumed as a declared dependency, but embedded or evolved as a forked codebase.

For researchers, the results suggest that fork-based vulnerability monitoring is an underexplored intersection of software supply chain security, mining software repositories, and security governance. Future work should investigate automated applicability analysis between upstream CVEs and downstream forks, better estimation of patch adaptation effort, and empirical measurement of how AppSec-mediated workflows affect remediation time and decision quality.

6. Related Work

Prior work has investigated fork-based development and the maintenance challenges created by long-lived divergence between upstream and downstream repositories. Salman [Salman 2021] discusses forks in social coding platforms, while Businge et al. [Businge et al. 2022] characterize reuse and maintenance practices among divergent forks. Sung et al. [Sung et al. 2020] show that integrating upstream changes into divergent forks may introduce conflicts and require manual resolution effort. Related studies on OSS reuse, such as CENTRIS [Woo et al. 2021], shows that reused open-source code is often modified, making provenance and inherited vulnerabilities harder to track. These works motivate our setting, but they do not focus on how security teams operationalize upstream vulnerability evidence to support remediation decisions in downstream forks.

Another line of research addresses vulnerable version identification and vulnerability management in OSS ecosystems. VERJava [Sun et al. 2022] and VERCATION [Cheng et al. 2026] proposes techniques to identify vulnerable versions of OSS components, while Ponta et al. [Ponta et al. 2020] study the detection, assessment, and mitigation of vulnerabilities in dependencies. More recently, studies on dependency automation, such as Dependabot-focused work [Mohayeji et al. 2025], have investigated how automated alerts and update mechanisms affect vulnerability mitigation. These approaches are relevant to software supply chain security [Williams et al. 2025], but they are mainly centered on dependency-level vulnerability management. In contrast, our work targets a downstream product derived from upstream repositories, where the threat of vulnerability and fix integration must be interpreted in the context of a modified fork.

Security patches and fix-commit mining have also received substantial attention. Li and Paxson [Li and Paxson 2017] conducted a large-scale empirical study of security patches, characterizing how vulnerabilities are fixed in open-source projects. CVEfixes [Bhandari et al. 2021] and MoreFixes [Akhoundali et al. 2024] provide large-scale datasets that connect CVEs to fixing commits, supporting data-driven vulnerability research. Hommersom et al. [Hommersom et al. 2024] investigate automated mapping between vulnerability advisories and fix commits, and VulCurator [Nguyen et al. 2022] focuses on detecting vulnerability-fixing commits. Our workflow builds on this research direction by using fix-commit metadata not as an end in itself, but as evidence for downstream maintenance decisions, including triage, patch adaptation, and risk acceptance.

Dependency-management tools such as Dependabot and conventional SCA approaches primarily focus on declared dependencies, affected versions, alerts, and automated updates. Vulnerable-version identification and fix-commit mining approaches, in turn, focus on identifying affected releases or connecting vulnerabilities to source-code changes. These approaches are complementary to our workflow. Our focus is the consolidation of vulnerability evidence for upstream repositories and supporting components

that may be embedded and modified as part of a divergent downstream fork. The resulting evidence is then interpreted by AppSec to support applicability assessment, patch adaptation, testing, and remediation planning. We did not conduct a controlled performance comparison; therefore, this distinction is functional rather than evidence of superior efficiency.

This paper differs from prior work by combining these perspectives in an applied government experience. Rather than proposing a general-purpose vulnerability dataset, dependency scanner, or fix-commit classifier, we present an automated monitoring workflow tailored to fork-based development. The workflow triangulates information from several sources to support AppSec-mediated decision-making among development, security testing, and DevSecOps teams. We evaluate this process in the context of a Brazilian government secure mobile communication app derived from Matrix/Element open-source clients, a setting in which upstream vulnerability monitoring is both a software supply chain concern and an organizational coordination problem.

7. Threats to Validity

External validity. This paper reports an experience from a single Brazilian government mobile communication project derived from Matrix/Element open-source clients. Thus, the findings should be interpreted as evidence from a specific security-sensitive fork-based context rather than as a general assessment of OSS-derived products.

The workflow's data sources and processing steps are not specific to the government domain. Therefore, it could potentially be instantiated in other security-sensitive sectors, such as healthcare or finance, provided that maintainers can identify the relevant upstream repositories and products and have an AppSec process for contextual applicability assessment. However, this transferability has not yet been empirically evaluated, and validation in additional projects remains future work.

Construct validity. The workflow relies on public vulnerability records, CPE mappings, GitHub advisories, CWE categories, and fixing-commit metadata to support upstream vulnerability monitoring. These artifacts are useful for security triage and remediation planning, but they do not automatically determine whether a vulnerability is present, reachable, enabled, or exploitable in a modified fork. Likewise, fix-commit size provides only an initial indication of patch scope and does not capture semantic complexity, architectural divergence, or regression risk. The workflow currently relies on CVSS and does not incorporate exploitation-likelihood or known-exploitation signals. Future work will evaluate EPSS and the CISA KEV catalog as complementary evidence for prioritization, without replacing downstream applicability analysis.

Internal validity. The analysis depends on the completeness and accuracy of public vulnerability metadata. NVD CPE mappings may be incomplete or imprecise, and relevant advisories may not always be linked to the expected CPE entries. We mitigate this by combining NVD queries with GitHub Advisory Database mining and CWE enrichment. However, missing metadata and AppSec expert judgment may still influence vulnerability classification, prioritization, and remediation decisions.

Replicability. The workflow was automated with Python scripts, and the resulting artifacts were organized into CSV files and notebooks. However, full replication may

be limited because vulnerability databases evolve over time and some downstream project details cannot be disclosed due to the government context. Thus, the data collection workflow is replicable, while the complete organizational decision process and applicability assessment are only partially reproducible.

8. Conclusion and Future Work

This paper reported a Brazilian government experience in monitoring upstream vulnerabilities for a fork-based secure mobile communication application derived from Matrix/Element open-source clients. We presented an automated workflow that correlates NVD, CPE, GitHub Advisory Database, CWE, and upstream fix-commit metadata to support downstream vulnerability triage and remediation planning.

Our results showed that the monitored upstream scope exposed 51 CVEs across seven projects, with most vulnerabilities classified as Medium severity, but also including High and Critical cases. The analysis also identified 44 CWE categories and 16 CVEs with available fixing-commit metadata, showing that upstream security exposure spans mobile clients, authentication components, cryptographic libraries, and protocol SDKs. These findings reinforce that vulnerability management in fork-based development is not only a technical detection activity, but also a coordination process that requires contextual assessment. The main lesson from this experience is that automated monitoring is most useful when it supports, rather than replaces, AppSec-mediated decision-making. By consolidating upstream vulnerability evidence, the workflow helped align development, security testing, and DevSecOps teams around decisions such as applying an upstream fix, adapting a patch, deferring analysis, or documenting risk acceptance.

Operationally, the workflow acts as an evidence layer between public vulnerability sources and downstream maintenance. Its outputs support AppSec teams in determining applicability, inspecting upstream patches, opening remediation tasks, requesting security validation, coordinating DevSecOps activities, monitoring deferred cases, and documenting risk decisions.

As future work, we plan to improve the automation of CVE-to-fork applicability analysis, estimate patch adaptation effort more precisely, and integrate the workflow into CI/CD security gates. We also intend to evaluate how this process affects remediation time, decision traceability, and the operational cost of maintaining security-sensitive fork-based applications.

References

Akhoundali, J., Nouri, S. R., Rietveld, K., and Gadyatskaya, O. (2024). Morefixes: A large-scale dataset of cve fix commits mined through enhanced repository discovery. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2024, page 42–51, New York, NY, USA. Association for Computing Machinery.

Alomari, H. W., Vendome, C., and Gyawali, H. (2025). A slicing-based approach for detecting and patching vulnerable code clones. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 60–72, Los Alamitos, CA, USA. IEEE Computer Society.

Bhandari, G., Naseer, A., and Moonen, L. (2021). Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, page 30–39, New York, NY, USA. Association for Computing Machinery.

Businge, J., Openja, M., Nadi, S., and Berger, T. (2022). Reuse and maintenance practices among divergent forks in three software ecosystems. *Empirical Softw. Engg.*, 27(2).

Cheng, Y., Zhang, T., Shar, L. K., Yang, S., Dong, C., Lo, D., Lv, S., Shi, Z., and Sun, L. (2026). Vercation: Precise vulnerable open-source software version identification based on static analysis and llm. *IEEE Transactions on Software Engineering*, 52(2):376–394.

Element (2023). Digital sovereignty is built on an open standard that enables federation. https://element.io/blog/digital-sovereignty-is-built-on-an-open-standard-that-enables-federation. Accessed: 2026-05-01.

Hommersom, D., Sabetta, A., Coppola, B., Nucci, D. D., and Tamburri, D. A. (2024). Automated mapping of vulnerability advisories onto their fix commits in open source repositories. *ACM Trans. Softw. Eng. Methodol.*, 33(5).

Imamura, D., Ishio, T., Kula, R. G., and Matsumoto, K. (2022). Bug-fix variants: Visualizing unique source code changes across github forks. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 157–161.

Li, F. and Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2201–2215, New York, NY, USA. Association for Computing Machinery.

Martins, J. A., Rego, P. A., de Macêdo, J. A., Silva, F. A., and Lagrota, V. (2026). Matrix protocol: a comprehensive systematic mapping study. *Journal of Cloud Computing*, 15(1):20.

Mohayeji, H., Agaronian, A., Constantinou, E., Zannone, N., and Serebrenik, A. (2025). Securing dependencies: A comprehensive study of dependabot's impact on vulnerability mitigation. *Empirical Softw. Engg.*, 30(3).

Nguyen, T. G., Le-Cong, T., Kang, H. J., Le, X.-B. D., and Lo, D. (2022). Vulcurator: a vulnerability-fixing commit detector. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1726–1730, New York, NY, USA. Association for Computing Machinery.

Ponta, S. E., Plate, H., and Sabetta, A. (2020). Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Softw. Engg.*, 25(5):3175–3215.

Salman, H. E. (2021). Feature-based insight for forks in social coding platforms. *Inf. Softw. Technol.*, 140(C).

Sun, Q., Xu, L., Xiao, Y., Li, F., Su, H., Liu, Y., Huang, H., and Huo, W. (2022). Verjava: Vulnerable version identification for java oss with a two-stage analysis. In *2022 IEEE*

International Conference on Software Maintenance and Evolution (ICSME), pages 329–339.

Sung, C., Lahiri, S. K., Kaufman, M., Choudhury, P., Wolk, J., and Wang, C. (2020). Towards understanding and fixing upstream merge induced conflicts in divergent forks: an industrial case study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 320–321, New York, NY, USA. Association for Computing Machinery.

Williams, L., Benedetti, G., Hamer, S., Paramitha, R., Rahman, I., Tamanna, M., Tystahl, G., Zahan, N., Morrison, P., Acar, Y., Cukier, M., Kästner, C., Kapravelos, A., Wermke, D., and Enck, W. (2025). Research directions in software supply chain security. *ACM Trans. Softw. Eng. Methodol.*, 34(5).

Woo, S., Park, S., Kim, S., Lee, H., and Oh, H. (2021). Centris: A precise and scalable approach for identifying modified open-source software reuse. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 860–872. IEEE Press.