Beneath the Mask: Can Contribution Data Unveil Malicious Personas in Open-Source Projects?

Author: Ruby Nealon, ruby@ruby.sh
Advisor: John Hally

Accepted: April 12, 2025

Abstract

In February 2024, after building trust over two years with project maintainers by making a significant volume of legitimate contributions, GitHub user "JiaT75" self-merged a version of the XZ Utils project containing a highly sophisticated well-disguised backdoor targeting *sshd* processes running on systems with the backdoored package installed. A month later, this package began to be distributed with popular Linux distributions until a Microsoft employee discovered the backdoor while investigating how a recent system upgrade impacted the performance of SSH authentication. Despite the potential impact this backdoor could have had globally, no tooling has been created for monitoring and identifying anomalous behavior by personas contributing to other open-source projects. This paper demonstrates how using graph databases and theory with OSINT contribution data gathered from GitHub can efficiently identify anomalous behaviors exhibited by the "JiaT75" persona across other open-source projects.

1. Introduction

On February 23, 2024, Version 5.6.0 of the XZ Utils project *libxz*/*liblzma* library package was released, introducing a backdoor targeting the *sshd* remote login server. Just over a month later, on March 29, it was discovered by Andres Freund, a software engineer at Microsoft, who first disclosed it to the Openwall mailing list. The backdoor was introduced to the project source code by a contributor known as Jia Tan (or "JiaT75"). JiaT75 had first started contributing to the project in January 2022, engaging in an incredibly sophisticated campaign of sock-puppetry. By the time the backdoored version was released, they were recognized as a project member with commit rights, publishing the backdoored version without prior review by the other maintainer (James, 2024).

The backdoor received much attention after the disclosure for reasons relating to the discovery of the backdoor, the backdoor itself, and the significant amount of time and effort the attacker had put into building trust with the "JiaT75" persona. In his own words, Freund, who disclosed the backdoor, did not consider himself a security researcher; instead, he discovered the backdoor whilst investigating high CPU usage on SSH logins, which he initially believed to be a performance regression (Freund, 2024).

The choice of the *sshd* package as a target was also interesting, as *sshd* does not depend directly or indirectly on the backdoored package. However, many major Linux distributions like Ubuntu and Debian use *systemd* for service management and distribute versions of packages like *sshd* patched for compatibility. As *systemd* depends on the XZ package, this caused the backdoored library to now be loaded with the *sshd* process indirectly (Freund, 2024).

For many, though, the most interesting part of the backdoor was the "JiaT75" persona and the level of trust over time the attacker had built by making significant legitimate-seeming contributions to the project – most of which are still present in the source code. Existing timelines and analyses like Tumbleson (2024) and Boehs (2024) demonstrate that the social engineering done by the attacker behind the backdoor likely extends far beyond the XZ Utils GitHub repository and even the specific "Jiat75" persona itself.

In the aftermath of the backdoor, work has already been done to implement technical controls against the weaknesses in how open-source package released and distributed that were exploited by the attacker. For example, “backseat-signed” is a tool created in response to the backdoor to assist with verifying the cryptographic chain-of-custody of Linux distribution packages and their upstream source code inputs (kpcyrd, n.d.). However, despite a lot of the social engineering identified in previously mentioned writing taking place within the structures of the GitHub and Git repositories themselves, there have been no attempts so far to automate detecting the anomalous behaviors exhibited by the JiaT75 persona in other open-source project repositories.

1.1. Graph Databases

Databases used by applications requiring data retrieval and storage are typically “relational”; data is modeled and stored in structured tables with rows and columns. This is the case for small databases aimed at personal and small business use, such as Microsoft Access, and commercial and non-commercial large-scale offerings like MySQL, Microsoft SQL Server, and Oracle. In contrast, graph databases model and store data as nodes connected by relationships (also referred to as edges or links). Though graph data can be modeled and stored in relational databases, graph databases are generally more performant for queries that traverse graph data. Graph database query languages are also considered more straightforward when writing queries against graph data (Vicknair et al., 2010).

Cybersecurity tools already use graph databases for use cases involving analyzing and investigating relationship-dense data. A well-known example of this is BloodHound. Using Active Directory objects like users and groups and their relationships like memberships and privileges, BloodHound uses graph theory to path find routes for privilege escalation from a given level of access in an environment, as well as the discovery of unusual or unexpected relationships (SpecterOps, n.d.).

The natural provenance inside Git repositories, spread across many projects and contributors, makes data from source forge websites like GitHub inherently relationship-dense. This creates an opportunity for using graph theory and graph databases to investigate and query open-source project contribution data.

2. Research Method

To validate the utility of signals from OSINT contributor data, public data will be gathered from XZ Utils and a control group of similar projects. The data will be collected from both GitHub and the GitHub-hosted Git repository itself, and ingested into a graph database. Behavior anomalous for open-source contributors identified in existing writing about the backdoor will be analysed, quantified and written as graph database queries. The queries will then be executed against the set of data collected from the scoped projects, and their results discussed for the suitability of the query in discovering behavior that may be worthy of further scrutiny.

2.1. Scoping

A total of 19 repositories hosted on GitHub.com containing the source code for direct and indirect, required, optional, and *make* dependency packages of *systemd* were selected from the Arch Linux package registry. The selected projects range from small (such as *hwdata* with only 414 commits with 14 unique GitHub contributors) to large (*libarchive* with 6640 commits across 307 contributors) with varying levels of organization and governance. The projects selected also have a varying level of "popularity" (presence across a sampled of opted-in Arch Linux installations) per the archlinux.de "Pkgstats" project (archlinux.de, n.d.)

GitHub repository owner/name	Arch repository /package name (popularity %)	# of commits in main branch	# of unique GitHub contributors
tukaani-project/xz	core/xz (100%)	2854	29
linux-audit/audit-userspace	core/audit (100%)	2725	62
p11-glue/p11-kit	core/libp11-kit (100%)	1474	82
seccomp/libseccomp	core/libseccomp (100%)	1047	66
vcrhonek/hwdata	core/hwdata (99.73%)	414	14
besser82/libxcrypt	core/libxcrypt (99.97%)	830	23
lz4/lz4	core/lz4 (100%)	3615	177
fukuchi/libqrencode	extra/qrencode (69.97%)	807	29
thom311/libnl	core/libnl (100%)	2138	143
tpm2-software/tpm2-tss	core/tpm2-tss (99.55%)	2905	119
libpwquality/libpwquality	extra/libpwquality (41.11%)	263	57

PJK/libcbor	extra/libcbor (44.29%)	1379	40
libexpat/libexpat	core/expat (100%)	4501	89
json-c/json-c	core/json-c (100%)	1376	139
libarchive/libarchive	core/libarchive (100%)	6640	307
PCRE2Project/pcre2	core/pcre2 (100%)	2064	58
eliben/pyelftools	extra/python-pyelftools (23.66%)	716	94
apjanke/ronn-ng	extra/ruby-ronn-ng (1.7%)	591	22
xkbcommon/libxkbcommon	extra/libxkbcommon (89.8%)	2510	66
shadow-maint/shadow	core/shadow (99.42%)	3865	177

Figure 1: Table of Projects in Scope for Collection (Data as of March 27, 2025)

The number of projects scoped was intentionally limited due to the hourly rate limits of the GitHub GraphQL API, with some single repositories requiring several hours to load all related data.

2.2. Tool development and environment

2.2.1. Environment

The graph database used for this experiment was Neo4j, also used by the BloodHound tool mentioned previously. The database and tooling used were run on a single macOS machine, with all data collected with GitHub GraphQL API and *git clone* calls.

2.2.2. Data Collection

To collect data efficiently and within the constraints of the aforementioned limits of the GitHub GraphQL API, automation was developed as a "gem" using the Ruby programming language. The automation for data collection first queries the total number of nodes for the desired connections of each repository (for example, pull requests or issues) and uses GraphQL variables to control the number of loaded per each and the cursor to load nodes after. This minimizes the number of requests and API quota used and ensures that only new, relevant data is loaded. Once all data has been loaded, the raw paginated connections are joined into arrays. The data is then serialized into JSON data files for flexibility in when and how data is ingested.

As the GraphQL API has a quota directly on the number of objects loaded per query at 5000 per hour, this can make loading all the commits for a repository and their authors/committers unrealistically expensive for larger repositories. In order to reduce the number of requests necessary, a “bare” *git clone* operation with appropriate filters is also performed for each repository to efficiently download the equivalent commit metadata without needing to enumerate commits through the API.

2.2.3. Data Ingestion

The same gem defines ORM models for nodes representing the different entities using ActiveGraph – a Ruby gem providing similar functionality to the popular ActiveRecord ORM distributed as a part of the Ruby on Rails web framework (neo4jrb, n.d.). A separate function provides functionality to traverse loaded data as a tree, upserting nodes based on the data available and tracking relationships based on the depth in the tree. Once all nodes are inserted, the relationships are reduced to the minimal number required, removing duplicates. They are then inserted in bulk.

After the data queried from the GitHub GraphQL API is ingested, the branch, commit, and author/committer identity data is ingested from the Git repository retrieved with the bare *git clone* call.

Finally, the minimum number of additional GitHub GraphQL API calls are made to resolve additional GitHub user data associated with email addresses used in the Git commits loaded.

A schema of node labels and relationships was defined via the ActiveGraph ORM to mirror the same objects and hierarchies found in each external data source. For example, the *HAS_GITHUB_PULL_REQUEST* relationship from *GithubRepository* to *GithubPullRequest* mirrors how pull requests are loaded for a repository in the API call.

The schema for all node labels and their defined external data sources is described in Figure 2 below.

Node label	Data source
GithubCommit	GitHub GraphQL API
GithubCommitComment	GitHub GraphQL API

GithubDiscussion	GitHub GraphQL API
GithubDiscussionComment	GitHub GraphQL API
GithubIssue	GitHub GraphQL API
GithubIssueComment	GitHub GraphQL API
GithubOrganization	GitHub GraphQL API
GithubPullRequest	GitHub GraphQL API
GithubPullRequestReview	GitHub GraphQL API
GithubPullRequestReviewComment	GitHub GraphQL API
GithubRepository	GitHub GraphQL API
GithubUser	GitHub GraphQL API
GithubUserContentEdit	GitHub GraphQL API
GitBranch	Git repository data from bare *git clone* call
GitCommit	Git repository data from bare *git clone* call
GitIdentity	Git repository data from bare *git clone* call

Figure 2: Table of Node Labels and Their External Data Sources

2.3. Defining Anomalous Criteria and Method of Investigation

2.3.1. Review of Existing Analyses and Writing

Existing analyses and articles covering the extended timeline leading up to the insertion and discovery of the XZ Utils backdoor cover social engineering efforts far beyond what would be detectable from just the GitHub and Git repository data. However, consistent patterns are identified regarding the activity of the JiaT75 user in the project.

The "JiaT75" GitHub user was created in January 2021, more than 3 years before the backdoor was created. However, the Git repository for XZ Utils dates back as far as 2007, and the project dates back even further. After creating their GitHub user, they immediately began contributing to other projects, including *libarchive* – one of the scoped repositories for data collection. In April 2022, the Jia Tan/JiaT75 persona begins regularly contributing to the XZ project, and in January 2023, they self-merge their first pull request.

While contributing to multiple projects and new maintainers joining projects is not unusual, the frequency and pacing of the JiaT75 persona are. Connor Tumbleson, a software engineer, highlighted this in his blog post covering the attack situation:

> So now I was curious when Jia Tan was created on GitHub and scrolled all the way back.

> …
>
> This user was not a very old account being only created in January of 2021 and one of their first public merged pull requests is also under investigation.
>
> I scroll the user's history on GitHub and I'm blown away - commits and fixes to projects all over - some seemingly great and normal while others being reverted as quick as possible. (Tumbleson, 2024)

The short time to become an active contributor and gain trust within the project is also noted. Evan Boehs, a software engineer also blogging about the attack, writes:

> Three days after the emails pressuring Lasse Collin to add another maintainer, JiaT75 makes their first commit to xz: Tests: Created tests for hardware functions... Since this commit, they become a regular contributor to xz (they are currently the second most active).
>
> …
>
> JiaT75 merges their first commit on January 7, 2023, which gives us a good indication of when they fully gain trust. (Boehs, 2024)

2.3.2. Quantifiable Anomalous Criteria

From these observations, several quantifiable factors are identified that, in combination, could suggest anomalous behavior:

- Project history age (the earliest and latest authored date of any commit in a project)
- Contribution activity within a project (the percentage of commits authored by a contributor in a project)
- Contribution history within a project (the earliest and latest dates of commits authored by the contributor)
- History of self-merging pull requests without review within a project (if any, the earliest merge request by a contributor merged without review by another party)

2.3.3. Defining the Sets of Anomalous Criteria To Investigate

This research will investigate two sets of criteria that indicate anomalous behavior: (1) contributors self-merging pull requests without review with limited project involvement and (2) contributors making an unusually significant share of all-time contributions for their total presence in the repository history.

Each set of criteria will be investigated by first writing and executing an unscoped query, analyzing the results and defining thresholds/filters where appropriate, then re-executing and discussing the set of criteria and results with regard to their suitability as a signal of potential social engineering. With the exception of the JiaT75 persona, contributor usernames will be anonymized in the results.

3. Findings and Discussion

3.1. Enriching Collected Data With Additional Relationships

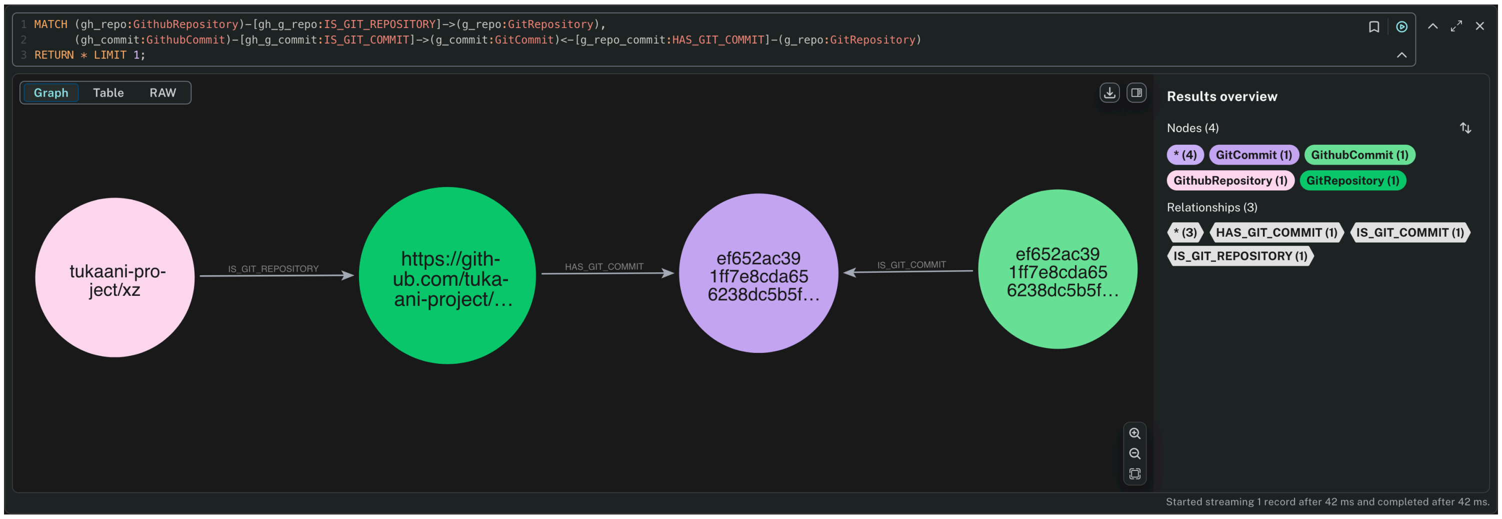
While the data collected from the GitHub GraphQL API and bare Git repository already have an inherent hierarchy that accurately describes the logical relationships between entities, like the repository having pull requests example mentioned earlier, other relationships are valuable for our investigation that do not exist in the upstream schema itself.

To assist with expressively writing queries to look for deep relationships in the data set, the following additional relationships were defined and populated for the ingested data.

3.1.1. IS_GIT_REPOSITORY and IS_GIT_COMMIT

These relationships link a *GithubRepository* or *GithubCommit* loaded from the GraphQL API to its corresponding *GitRepository* or *GitCommit* loaded from the bare Git repository data.

An example query demonstrating these relationships ran against the scoped data set is shown in Figure 3.



```
MATCH (gh_repo:GithubRepository)-[gh_g_repo:IS_GIT_REPOSITORY]->(g_repo:GitRepository),
      (gh_commit:GithubCommit)-[gh_g_commit:IS_GIT_COMMIT]->(g_commit:GitCommit)<-[g_repo_commit:HAS_GIT_COMMIT]-(g_repo:GitRepository)
RETURN * LIMIT 1;
```

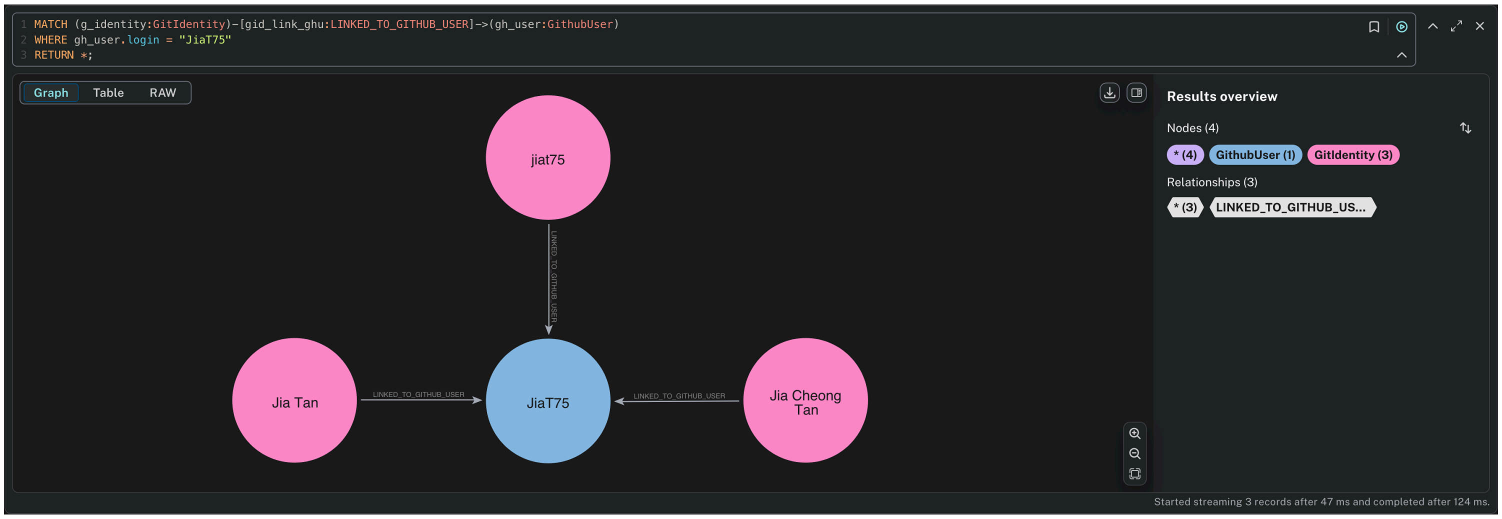
Started streaming 1 record after 42 ms and completed after 42 ms.

Figure 3: Query Demonstrating *IS_GIT_REPOSITORY* and *IS_GIT_COMMIT*

3.1.2. LINKED_TO_GITHUB_USER

This relationship links a *GitIdentity* to a *GithubUser* from the set of commits collected for a repository from the GraphQL API. As GitHub links a commit to a user by the email address, only a single commit needs to be loaded from the GraphQL API, where the email address is the committer or author to create this relationship between the resulting *GithubUser* node and all *GitIdentity* nodes with the same email address.

An example query demonstrating this relationship by loading the *GitIdentity* nodes linked to the *GithubUser* node for “JiaT75” is shown in Figure 4.



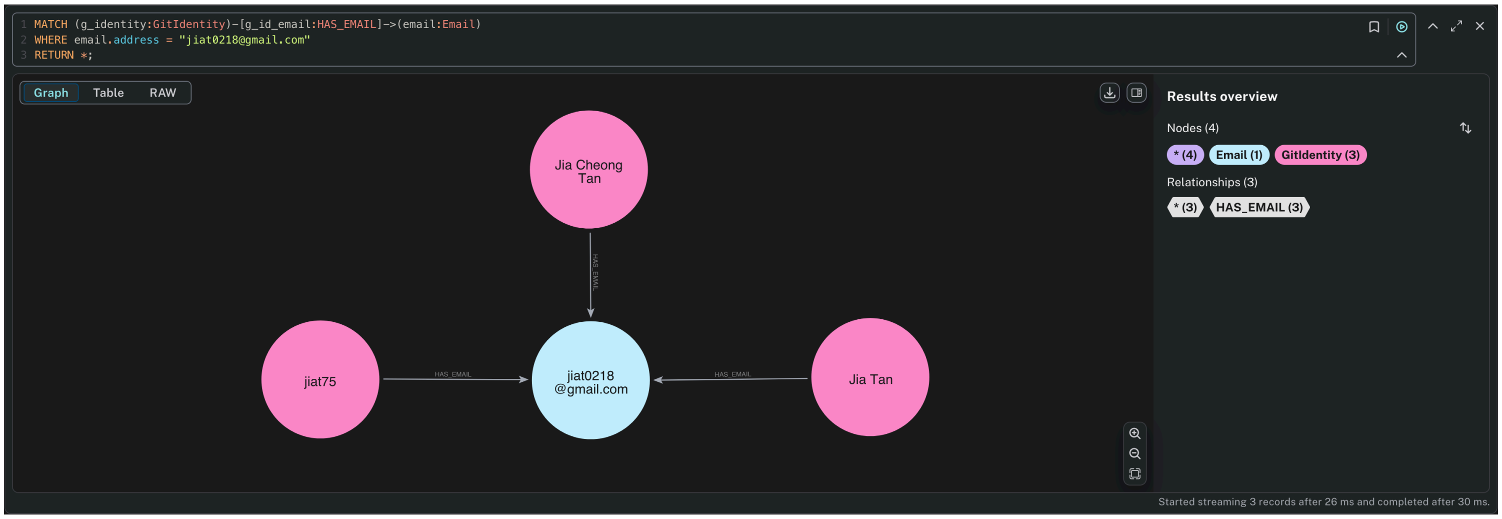
```
MATCH (g_identity:GitIdentity)-[gid_link_ghu:LINKED_TO_GITHUB_USER]->(gh_user:GithubUser)
WHERE gh_user.login = "JiaT75"
RETURN *;
```

Figure 4: Query Demonstrating *LINKED_TO_GITHUB_USER*

3.1.3. HAS_EMAIL

This relationship links a GithubUser, GithubOrganization, or GitIdentity to a new *Email* node created for each unique email address discovered.

An example query demonstrating this relationship by loading the *GitIdentity* nodes linked to the *Email* node for the email address “jiat0218@gmail.com” is shown in Figure 5.



```
MATCH (g_identity:GitIdentity)-[g_id_email:HAS_EMAIL]->(email:Email)
WHERE email.address = "jiat0218@gmail.com"
RETURN *;
```

Started streaming 3 records after 26 ms and completed after 30 ms.

Figure 5: Query Demonstrating *HAS_EMAIL*

3.2. Querying for Unreviewed, Self-Merged Pull Requests

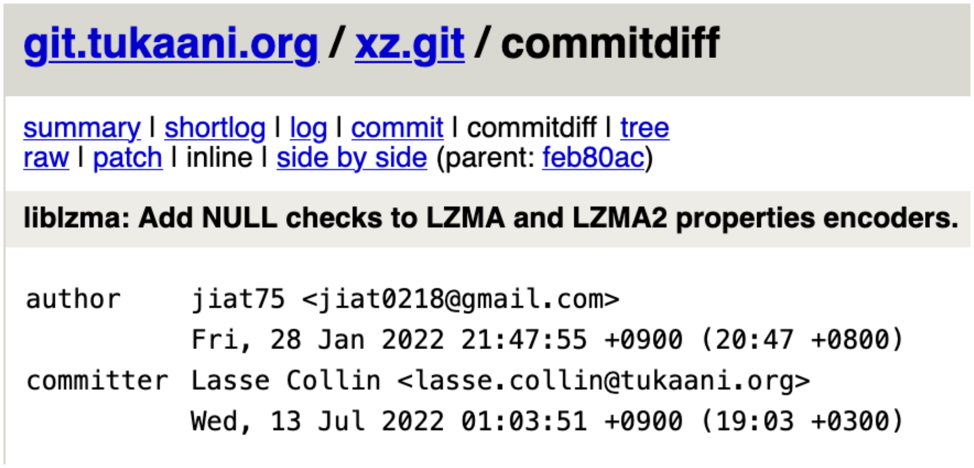
3.2.1. Query Creation

The first candidate criteria set to investigate earlier was a contributor's history of unreviewed (by another party), self-merged pull requests in relation to the age of the repository and when they first began contributing to it. To effectively measure this, a query was prepared collecting the following values for each *GithubUser* in each *GithubRepository* where they had self-merged at least one pull request:

- The *GitHubUser* username, creation timestamp, first authored commit timestamp, and all associated email addresses
- The *GithubRepository* name with its owner, creation timestamp, and first-authored commit timestamp
- The URL of and merge timestamp of the first self-merged pull request
- The total number of self-merged pull requests
- The difference between the *GithubUser* and *GithubRepository* first authored commit timestamps (i.e., the repository age at the time of the first contribution)
- The difference between the *GithubUser*'s first authored commit timestamp and the merge timestamp of the first self-merged pull request

This query returned 40 users, approximately 2% of the 1996 unique users in the data set who had authored a pull request. No user in the data set had self-merged an unreviewed pull request in more than one repository.

The data returned for the JiaT75 persona differs slightly from what Boehs (2024) stated in his timeline but was confirmed as correct by cross-referencing against GitHub and the Git repository; the first authored commit by JiaT75 in the repository was in late January 2022 (though not *committed* until July), and their first self merged pull-request was in December of the same year. GitHub does not show separate dates for authored and committed times of commits, so this could have caused a misinterpretation.

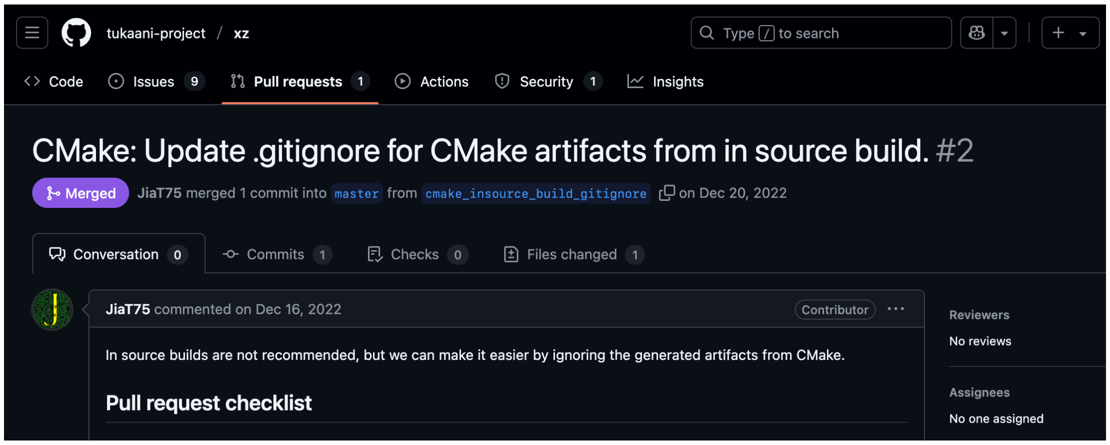

git.tukaani.org / xz.git / commitdiff

summary | shortlog | log | commit | commitdiff | tree
raw | patch | inline | side by side (parent: feb80ac)

liblzma: Add NULL checks to LZMA and LZMA2 properties encoders.

```
author     jiat75 <jiat0218@gmail.com>
           Fri, 28 Jan 2022 21:47:55 +0900 (20:47 +0800)
committer  Lasse Collin <lasse.collin@tukaani.org>
           Wed, 13 Jul 2022 01:03:51 +0900 (19:03 +0300)
```

Figure 6: JiaT75's First Authored Commit to XZ Utils



CMake: Update .gitignore for CMake artifacts from in source build. #2

Merged JiaT75 merged 1 commit into master from cmake_insource_build_gitignore on Dec 20, 2022

JiaT75 commented on Dec 16, 2022 Contributor

In source builds are not recommended, but we can make it easier by ignoring the generated artifacts from CMake.

Pull request checklist

Reviewers
No reviews

Assignees
No one assigned

Figure 7: JiaT75's First Unreviewed, Self-Merged Pull Request

The correct contributor age for JiaT75, confirmed from the dates in Figures 6 and 7, returned in the results, is 10 months. Nearly half of the results (19) have a contributor age of less than 24 months, which was used as a threshold for the final query.

The query was then further iterated, adding exclusions for irrelevant (i.e., now long-term contributors) and edge cases:

- Excluded results where the user's first authored Git commit was more than 4 years ago – reduces results from 19 to 7 records
- Excluded results where the contributor age was less than or equal to zero – reduces results from 7 records to 4 records

3.2.2. Discussion of Results

Figure 8 presents the relevant data from the results after executing the final query, provided in Appendix A.

Name	Repository	Contributor age at first self-merge	# of unreviewed, self-merged pull requests
Contributor A	PCRE2Project/pcre2	2 months	43
Contributor B	PCRE2Project/pcre2	7 months	7
JiaT75	tukaani-project/xz	10 months	24
Contributor C	linux-audit/audit-userspace	20 months	3

Figure 8: Results of the Unreviewed, Self-Merged Pull Request Query

The circumstances for each contributor in the results were investigated and are described briefly below:

- Contributor A had recently started actively contributing to the project and became the principal maintainer after an in-person meeting with the former maintainer. Contributor A has previously made commits with a corporate email address for a large American technology company and was verified as a current employee on the company's website.
- Contributor B was a long-term contributor to the project before its source code was hosted on GitHub. However, the email address on the Git

identity used for their earlier contributions is not linked to their current GitHub user.

- Contributor C uses a corporate email address for another sizeable American technology company on their commits and has the same domain name as the project maintainer listed in the AUTHORS file.

Though none of the additional results were true positives, including the JiaT75 user and the overall limited number of matches proportional to the total number of users assessed (approximately 0.2%), the query is suitable for detecting potential social engineering cases. Additionally, the discovery of earlier commits by the JiaT75 user suggests there is utility in using similar queries to investigate attacks that have already occurred.

3.3. Querying for Significant Contribution Relative to Presence

3.3.1. Query Creation

The second candidate criteria set to investigate earlier was a contributor making a significant share of all-time contributions relative to a contributor's presence in the repository history.

As was done for the first candidate criteria, a query was prepared to collect the following values for each *GithubUser* in each *GithubRepository* where they had made at least one commit:

- The *GitHubUser* username, creation timestamp, first authored commit timestamp, and all associated email addresses
- The *GithubRepository* name with its owner, creation timestamp, and first-authored commit timestamp
- The difference between the *GithubUser* and *GithubRepository* first authored commit timestamps (i.e., the repository age at the time of the first contribution)
- The difference between the current date and the *GithubRepository* first authored commit timestamp (i.e., the repository age)
- The percentage of commits authored by the user in the Git repository

- The difference between the first and last user-authored commit timestamps, as a percentage of the difference of the first and last authored commit timestamps for the repository (i.e., what percentage of the repository history the contributor has been present)

This query returned 2023 unique combinations of users with contributions in repositories. The data for JiaT75 indicates they authored 17.4% of commits to the repository, despite their contributions only taking place over 12.5% of the repository's history. As the intent of the role is to catch potential social engineering efforts before they occur, looser thresholds were chosen, matching contributors who have authored more than 5% of commits with less than 20% presence in the repository. Rerunning the query with the threshold returned six unique contributors, and after applying the same exclusion for the first-authored over four years ago, reduced it to 4.

3.3.2. Discussion of Results

Figure 9 presents the relevant data from the results after executing the final query, provided in Appendix B.

Name	Repository	% of authored commits	% of time presence in the commit history
Contributor A	PCRE2Project/pcre2	6.91%	4.66%
JiaT75	tukaani-project/xz	17.38%	12.47%
Contributor D	p11-glue/p11-kit	7.18%	16.67%
Contributor E	shadow-maint/shadow	18.4%	18.28%

Figure 9: Results of the Significant Contribution Relative to Presence Query

Again, the circumstances of each additional contributor were investigated, and are briefly described below:

- Contributor A also appeared in the results of the other query and was previously discussed.

- Contributor D uses a corporate email address for the same domain name as Contributors B and C. They are also listed as a maintainer in the project README.md file.
- Contributor E uses an email address with a domain associated with a sizeable open-source software project and was verified as a long-term contributor through information published on the project's website. They are also listed as a project maintainer in the AUTHORS file.

Though once again, the results did not contain any new true positives, the small number of results with a relatively loose threshold relative to what would have been observable for the JiaT75 persona again support this being a viable signal to investigate potential social engineering.

4. Recommendations and Implications

4.1. Recommendations

This research proves that social engineering attacks against open-source projects can potentially be detected via analysis of contributor metadata. Security practitioners looking to monitor for or investigate social engineering attacks against open-source projects should use automation where possible to collect and analyze contribution data from project contributors.

4.2. Non-recommendations

As was observed with the case of Contributor A, despite the aggregated data signaling suspicious or otherwise unusual behavior, upon investigation, it was discovered that they had been vetted by the former maintainer and recently taken over the responsibilities of the project.

Legitimate reasons exist for open-source contributors to use personas/aliases, or otherwise have a limited web presence outside of their open-source contributions. Security practitioners and project maintainers responding to signals raised by this kind of analysis should not assume malicious behavior while investigating. While it is encouraged to be conscious and apply caution when granting privileges or transferring responsibilities to new parties, such caution should not dissuade or hinder legitimate contributors from participating.

4.3. Future Research

This research validates the potential for investigation, but only on a smaller sample set of projects at a particular point in time. Potential future research could encompass one or more of the following elements:

- Continuous collection and ingestion of data.
- Deeper traversal/further collection of data from the GitHub GraphQL API.
- Additional data sources like the *gharchive* project should be used to minimize the number of requests necessary for the GitHub GraphQL API.
- Collection of data from other source code forges like GitLab.com or Gitea.com, as well as project-managed self-hosted instances.
- Collection of data from mailing lists relevant to the project (i.e., project mailing list, Linux distribution mailing lists relevant to the project) and identification of similar usage of sock puppets to pressure project maintainers, as was suspected of the XZ Utils backdoor attacker.
- Collection of other web presences for monitored contributing personas (i.e., LinkedIn or other social media profiles).
- Use of LLM to identify major events like transferring maintainership in project issue trackers and mailing lists.
- Use of sentiment analysis to identify unusual pressure applied towards maintainers in project issue trackers and mailing lists.

5. Conclusion

This research demonstrated the potential of using anomalous contribution patterns in open-source projects as a signal of social engineering attacks like the XZ Utils backdoor. Both sets of criteria tested against aggregate contributor data, looking at the timescale they gain privileged access to or become significant participants, show viability as good signals for similar projects. Further work to continuously monitor projects and collect additional data sources is encouraged and could raise an early warning for a similar attack in the future.

References

James, S. (2024, March 30). *FAQ on the xz-utils backdoor (CVE-2024-3094)*. GitHub Gist. Retrieved February 2, 2025, from https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27

Freund, A. (2024, March 29). *Oss-security – backdoor in upstream xz/liblzma leading to ssh server compromise*. Openwall. Retrieved February 2, 2025, from https://www.openwall.com/lists/oss-security/2024/03/29/4

neo4jrb. (n.d.). activegraph (README.md). GitHub. Retrieved March 25, 2025, from https://github.com/neo4jrb/activegraph

Tumbleson, C. (2024, March 31). *Watching xz unfold from afar*. Ramblings of a Tampa engineer. Retrieved March 29, 2025, from https://connortumbleson.com/2024/03/31/watching-xz-unfold-from-afar/

Boehs, E. (2024, April 8). *Everything I Know About the XZ Backdoor*. boehs.org. Retrieved March 29, 2025, from https://boehs.org/node/everything-i-know-about-the-xz-backdoor

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010). A comparison of a graph database and a relational database: a data provenance perspective. *Proceedings of the 48th Annual ACM Southeast Conference*. Presented at the Oxford, Mississippi. Doi:10.1145/1900008.1900067

SpecterOps. (n.d.). *Bloodhound – Six Degrees of Domain Admin (README.md)*. GitHub. Retrieved March 25, 2025, from https://github.com/SpecterOps/BloodHound

kpcyrd. (n.d.). backseat-signed (README.md). GitHub. Retrieved March 25, 2025, from https://github.com/kpcyrd/backseat-signed archlinux.de. (n.d.). *Package statistics & comparisons*. pkgstats.archlinux.de. Retrieved March 27, 2025, from https://pkgstats.archlinux.de/packages

Appendix A
Cypher Query for Unreviewed, Self-Merged Pull Requests

```
MATCH (gh_user:GithubUser)<-[:AUTHORED_BY_GITHUB_USER]-
(gh_pr:GithubPullRequest)-[:MERGED_BY_GITHUB_USER]->(gh_user),
(gh_pr)<-[:HAS_GITHUB_PULL_REQUEST]-(gh_repo:GithubRepository)
WHERE gh_repo.is_fork = false AND gh_pr.merged = true
OPTIONAL MATCH (gh_pr)-[:HAS_GITHUB_PULL_REQUEST_REVIEW]-
>(gh_pr_review:GithubPullRequestReview)-[:AUTHORED_BY_GITHUB_USER]-
>(gh_pr_review_user:GithubUser)
WHERE gh_pr_review_user <> gh_user AND gh_pr_review.state = "APPROVED"
WITH gh_user, gh_repo, gh_pr, collect(gh_pr_review_user) AS gh_pr_review_users
MATCH (gh_repo)-[:IS_GIT_REPOSITORY]->(git_repo:GitRepository)-
[:HAS_GIT_COMMIT]->(repo_git_commit:GitCommit)
WITH gh_user, gh_repo, gh_pr, gh_pr_review_users, git_repo,
min(repo_git_commit.authored_at) AS repo_earliest_git_commit_authored_at
WHERE size(gh_pr_review_users) = 0
MATCH (git_repo)-[:HAS_GIT_COMMIT]->(user_git_commit:GitCommit)<-
[:AUTHORED_GIT_COMMIT]-(:GitIdentity)-[:LINKED_TO_GITHUB_USER]->(gh_user)
WITH gh_user, gh_repo, gh_pr, repo_earliest_git_commit_authored_at,
min(user_git_commit.authored_at) AS user_earliest_git_commit_authored_at
MATCH (git_identity:GitIdentity)-[:LINKED_TO_GITHUB_USER]->(gh_user)
WITH gh_user, gh_repo, gh_pr, repo_earliest_git_commit_authored_at,
user_earliest_git_commit_authored_at, collect( DISTINCT git_identity.email) AS
user_identity_emails
ORDER BY gh_pr.merged_at ASC
WITH gh_user, gh_repo, repo_earliest_git_commit_authored_at,
user_earliest_git_commit_authored_at, user_identity_emails, count(gh_pr) AS
user_self_merged_pr_count, collect(gh_pr)[0] AS first_user_self_merged_pr,
(duration.inMonths(datetime({epochSeconds:
user_earliest_git_commit_authored_at}), datetime({epochSeconds:
collect(gh_pr)[0].merged_at}))).months) AS contributor_age_at_first_self_merge
WHERE contributor_age_at_first_self_merge <= 24 AND
user_earliest_git_commit_authored_at >= (datetime() - duration({years:
4})).epochSeconds AND contributor_age_at_first_self_merge > 0
RETURN gh_user.login AS github_user_login,
datetime({epochSeconds: gh_user.created_at}) AS github_user_created_date,
user_identity_emails,
gh_repo.name_with_owner AS github_repository_name_with_owner,
(duration.inMonths(datetime({epochSeconds:
repo_earliest_git_commit_authored_at}), datetime({epochSeconds:
user_earliest_git_commit_authored_at})).months / 12.0) AS
repo_age_at_user_first_contribution,
contributor_age_at_first_self_merge,
user_self_merged_pr_count,
first_user_self_merged_pr.url AS first_user_self_merged_pr_url,
datetime({epochSeconds: repo_earliest_git_commit_authored_at}) AS
repo_earliest_git_commit_authored_at,
datetime({epochSeconds: gh_repo.created_at}) AS github_repo_created_at,
datetime({epochSeconds: user_earliest_git_commit_authored_at}) AS
user_earliest_git_commit_authored_at,
datetime({epochSeconds: first_user_self_merged_pr.merged_at}) AS
first_user_self_merged_pull_request_merged_at
ORDER BY contributor_age_at_first_self_merge ASC;
```

Appendix B Cypher Query for Significant Contribution Relative to Presence

```
MATCH (gh_repo:GithubRepository)-[:IS_GIT_REPOSITORY]->(git_repo:GitRepository)
WHERE gh_repo.is_fork = false
MATCH (git_repo)-[:HAS_GIT_COMMIT]->(git_commit:GitCommit)
WITH gh_repo, git_repo, count(git_commit) AS total_git_commits,
min(git_commit.authored_at) AS repo_earliest_git_commit_authored_at,
max(git_commit.authored_at) AS repo_latest_git_commit_authored_at
MATCH (git_repo)-[:HAS_GIT_COMMIT]->(user_git_commit:GitCommit)<-
[:AUTHORED_GIT_COMMIT]-(git_identity:GitIdentity)-[:LINKED_TO_GITHUB_USER]-
>(gh_user:GithubUser)
WITH gh_user, gh_repo, repo_earliest_git_commit_authored_at,
repo_latest_git_commit_authored_at, total_git_commits,
min(user_git_commit.authored_at) AS user_earliest_git_commit_authored_at,
max(user_git_commit.authored_at) AS user_latest_git_commit_authored_at,
((count( DISTINCT user_git_commit) * 1.0 / total_git_commits) * 100) AS
commit_percentage
MATCH (git_identity:GitIdentity)-[:LINKED_TO_GITHUB_USER]->(gh_user)
WITH gh_user, gh_repo, repo_earliest_git_commit_authored_at,
user_earliest_git_commit_authored_at, repo_latest_git_commit_authored_at,
user_latest_git_commit_authored_at, commit_percentage, collect( DISTINCT
git_identity.email) AS user_identity_emails,
(duration.inMonths(datetime({epochSeconds:
repo_earliest_git_commit_authored_at}), datetime({epochSeconds:
user_earliest_git_commit_authored_at})).months / 12.0) AS
repo_age_at_user_first_contribution,
(duration.inMonths(datetime({epochSeconds:
repo_earliest_git_commit_authored_at}), datetime({epochSeconds:
datetime().epochSeconds})).months / 12.0) AS repo_current_age,
((duration.inSeconds(datetime({epochSeconds:
user_earliest_git_commit_authored_at}), datetime({epochSeconds:
user_latest_git_commit_authored_at})).seconds /
toFloat(duration.inSeconds(datetime({epochSeconds:
repo_earliest_git_commit_authored_at}), datetime({epochSeconds:
repo_latest_git_commit_authored_at})).seconds)) * 100) AS presence_percentage
WHERE commit_percentage > 5 AND presence_percentage < 20 AND
user_earliest_git_commit_authored_at >= (datetime() - duration({years:
4})).epochSeconds
RETURN gh_user.login AS github_user_login,
datetime({epochSeconds: gh_user.created_at}) AS github_user_created_date,
user_identity_emails AS user_identity_emails,
gh_repo.name_with_owner AS github_repository_name_with_owner,
commit_percentage,
presence_percentage,
repo_age_at_user_first_contribution,
repo_current_age,
datetime({epochSeconds: repo_earliest_git_commit_authored_at}) AS
earliest_repository_commit_date,
datetime({epochSeconds: user_earliest_git_commit_authored_at}) AS
earliest_user_commit_date,
datetime({epochSeconds: repo_latest_git_commit_authored_at}) AS
latest_repository_commit_date,
datetime({epochSeconds: user_latest_git_commit_authored_at}) AS
latest_user_commit_date
ORDER BY presence_percentage ASC;
```