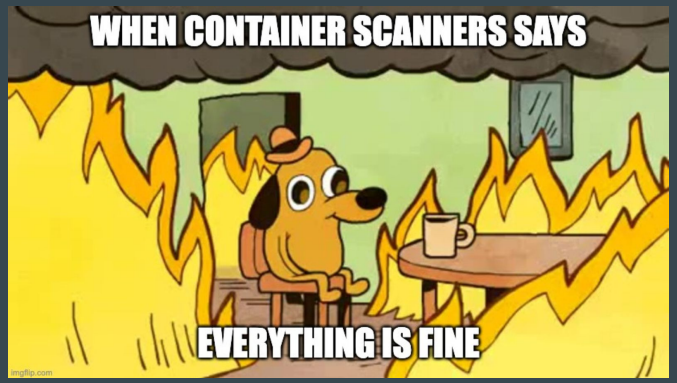
Smart & Distributed Vulnerability Remediation for Container Image ...

Xinzhe Jiang, Jingyi Huang, Pingcheng Dong, Yinan An

Introduction

Motivation:

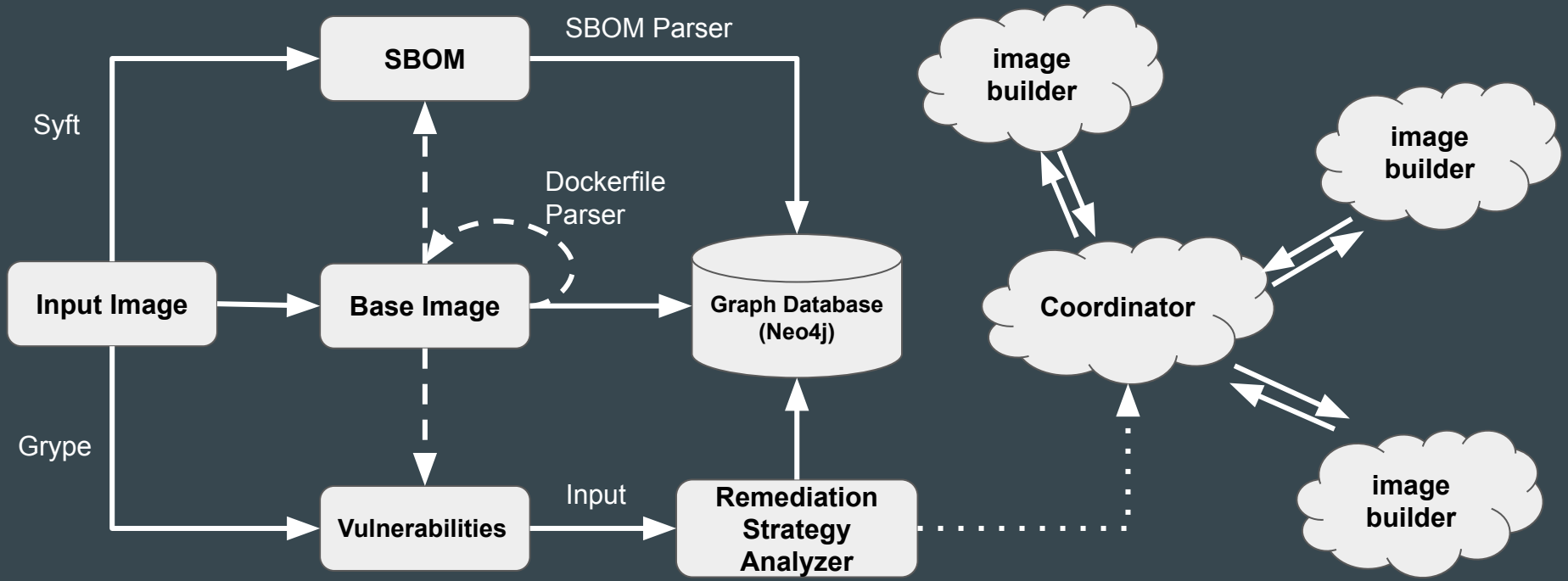
As the patterns for container building become layered, the traditional ways of vulnerability remediations are becoming very inefficient. We need a new way to find and repair the vulnerability.



Solution:

In this project, we designed a system that would allow us to optimize remediation workflows. There is also a graph database to capture dependency across images.

Project Architecture



Dockerfile Crawler

In order to give the user a better experience, we've added a simply crawler. The user can simply give the name of the base image and it will download the corresponding dockerfile from the dockerhub.

Dockerfile Parser

Takes in a dockerfile and returns a base image with its type. This process get ready all the information needed by SBOM.

Sample output:

```
type Image struct {
        Name      string   json:"name"
        Tag       string    json:"tag"
        OSName    string    json:"os_name"
        OSVersion string    json:"os_version"
        SHA256    string    json:"sha256"
        Metadata  string    json:"metadata"
        Packages  []Package json:"packages"
        Scanned   bool      json:"scanned"
}
```

Log4j Vulnerability

Log4Shell (CVE 2021-44228)

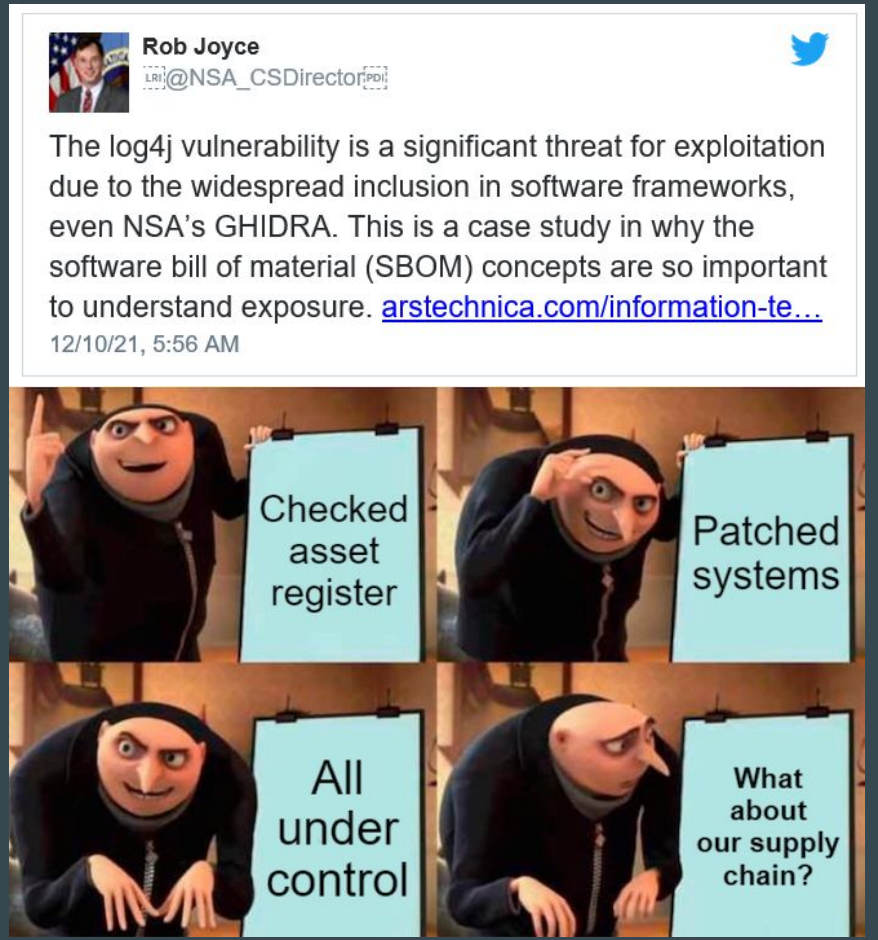
- Vulnerability in Apache Log4j, commonly used logging utility in Java application
- JNDI Injection: Remotely run code and gain access to all data on the affected machine
- Other exploitations: Ransomware

Software Supply Chain & SBOM

Software Bill of Material (SBOM):

Stores software artifact components and metadata

- Licensing information
- List of dependencies
- Persistent references
- Other auxiliary information



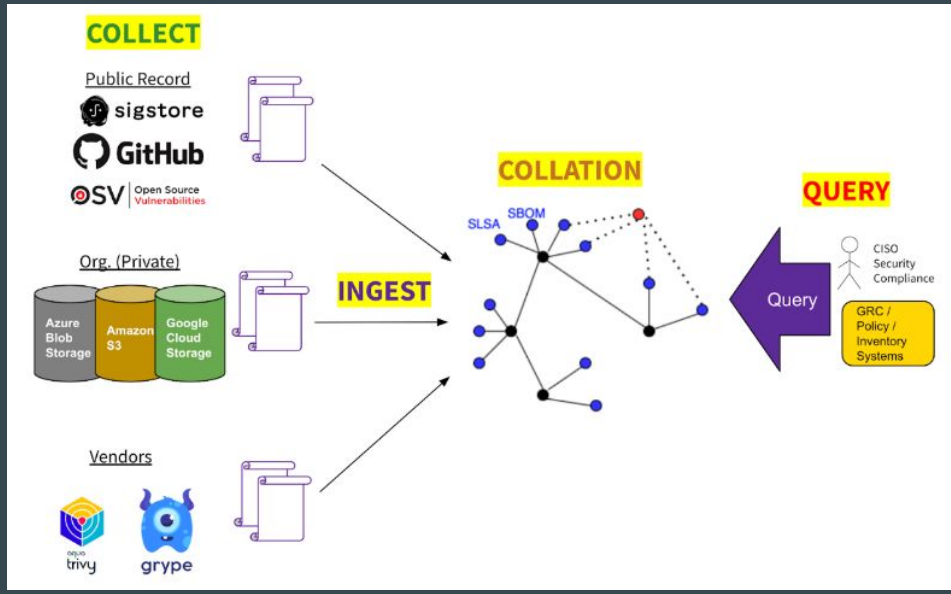
SBOM Generator & Parser

1. Execute Syft command to generate SBOM in SPDX JSON format
2. Parse SBOM file and generate dependency graph for the input images

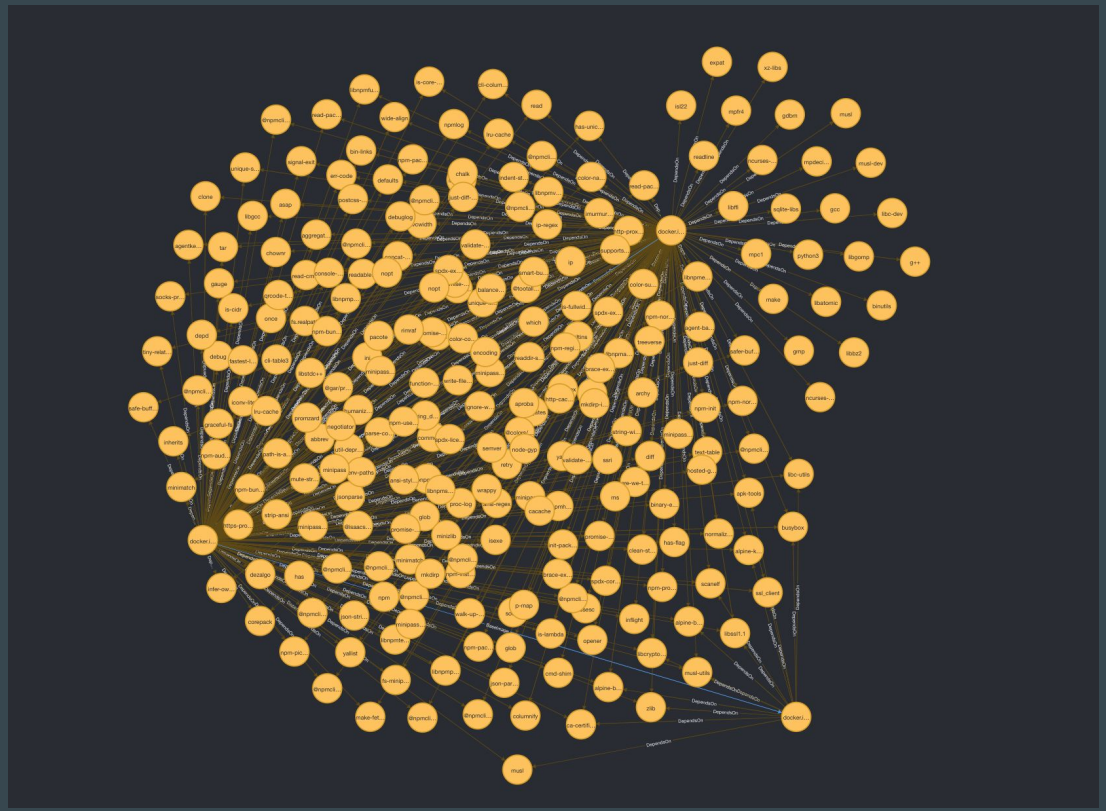
GUAC - Graph for Understanding Artifact Composition

GUAC has four major areas of functionality:

- **Collection**: collects software metadata from various source
- **Ingestion**: import data on artifacts, vulnerabilities, projects...
- **Collation**: traverse dependency and assemble to dependency tree
- **Query**: query the assembled graph for metadata attached to, or related to, entities within the graph



Dependency Graph



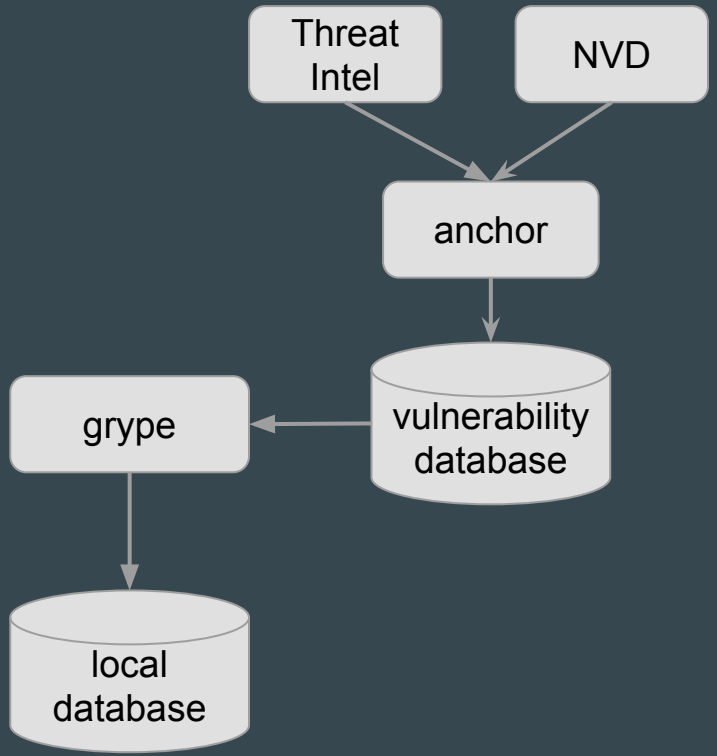
Grype: Vulnerability Scanning

When Grype performs a scan for vulnerabilities, it does so using a vulnerability database that's stored on your local filesystem, which is constructed by pulling data from a variety of publicly available vulnerability data sources

sources include:

- Alpine Linux SecDB
- Amazon Linux ALAS
- RedHat RHSAs
- Debian Linux CVE Tracker
- Github GHSAs
- National Vulnerability Database (NVD)
- Oracle Linux OVAL
- RedHat Linux Security Data
- Suse Linux OVAL
- Ubuntu Linux Security

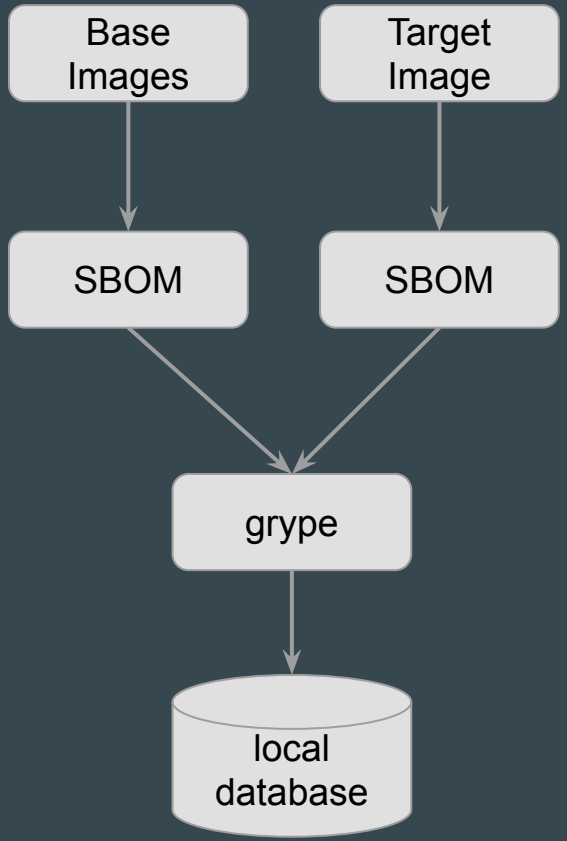
Grype checks for new updates to the vulnerability database to make sure that every scan uses up-to-date vulnerability information



Grype: Vulnerability Scanning

Grype in our project

1. Get SBOM of Base Image and Target Image
2. Grype scan for vulnerabilities using local database



Generating Remediation Workflow

Generating a directed graph(tree like) structure for base image relations

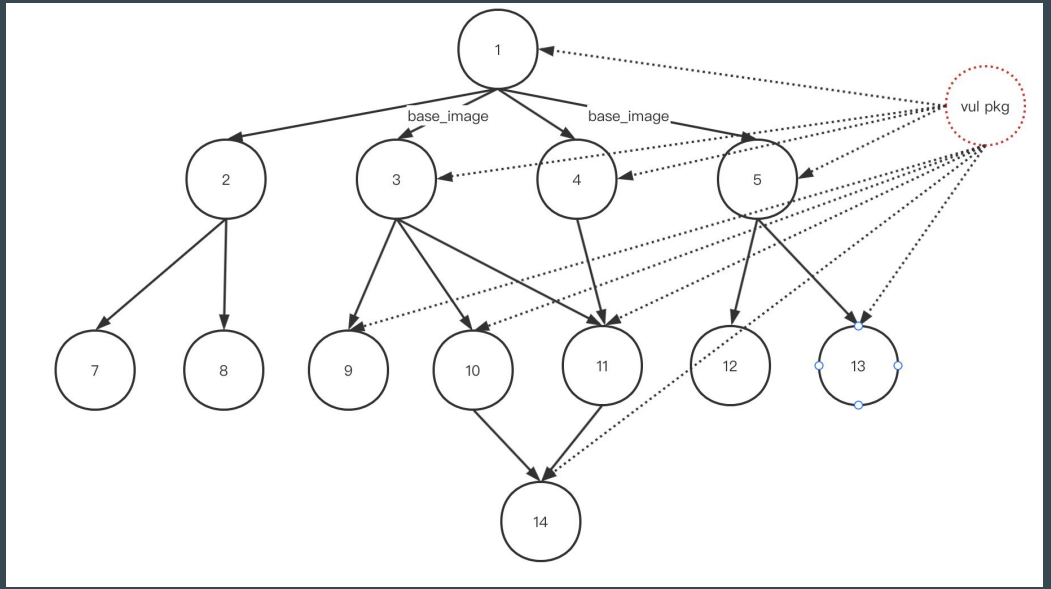
1. Query for all related images

2. DFS traversal

- decide fix/rebuild

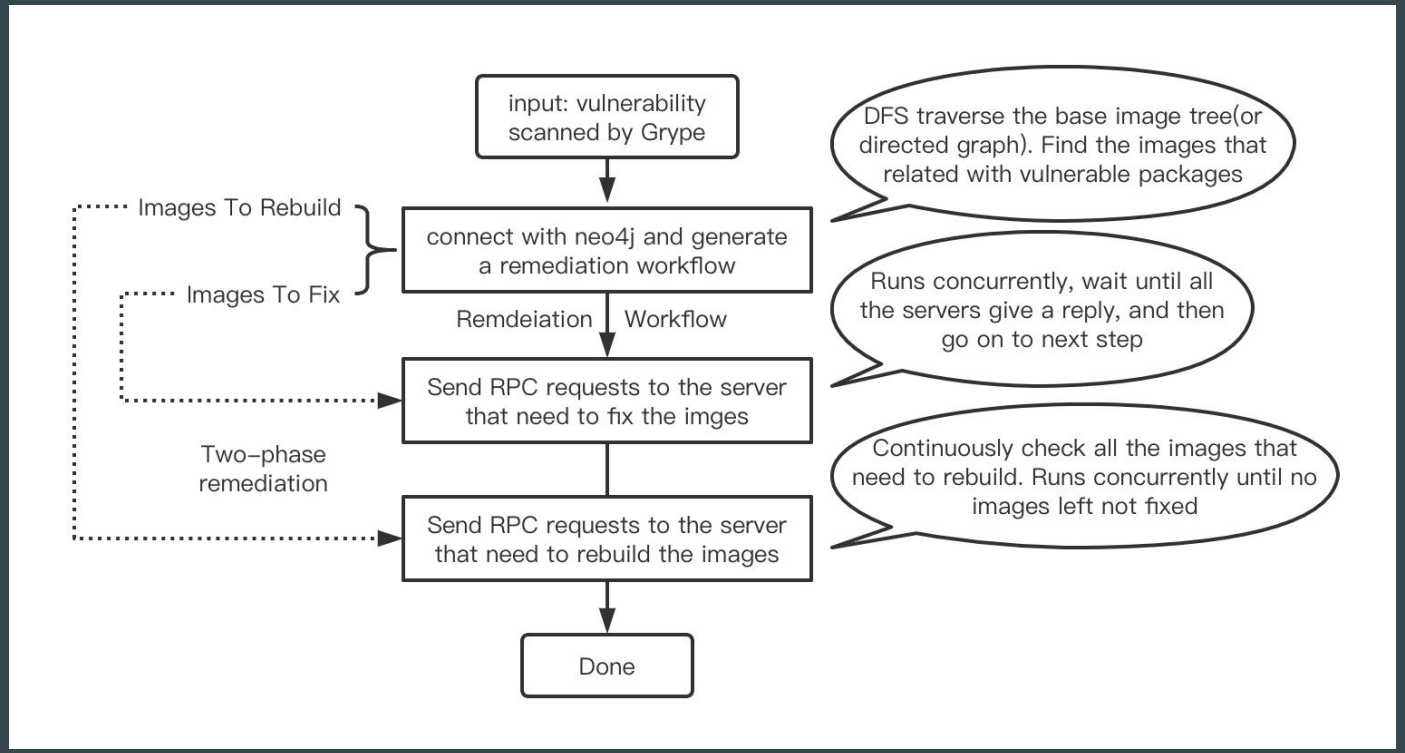
3. return the images that need to fix

and the images that need to rebuild



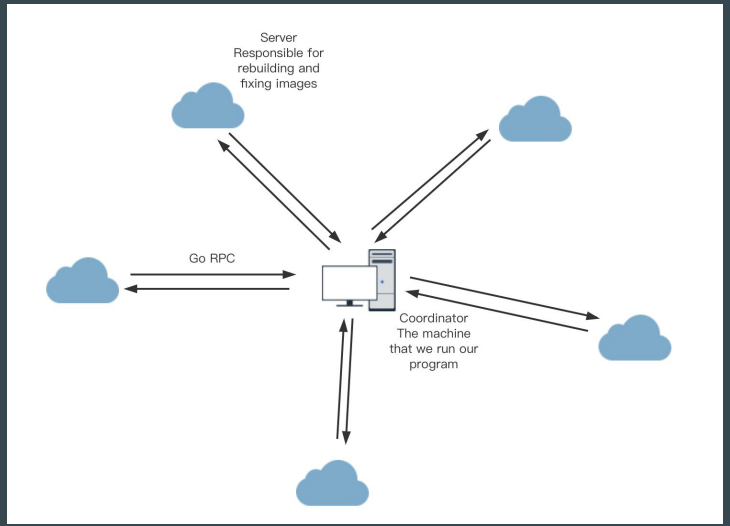
Fix 9, 14, 13 Rebuild 10, 11, 3, 4, 5, 1

Two-step distributed remediation



Concurrency Control and RPC

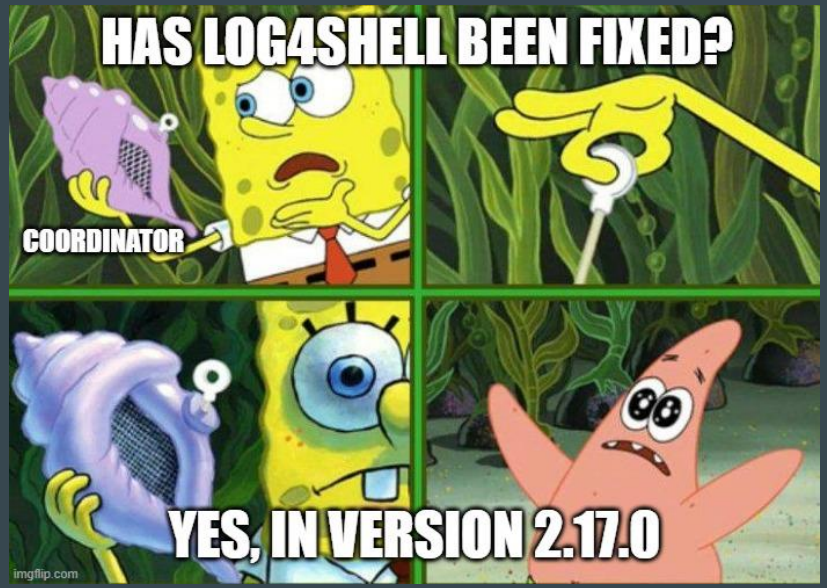
Servers' addresses of image builder are stored as configuration files on coordinator. Need to establish connection and use RPC to control different servers to fix/rebuild images.



Concurrency Control and RPC

RequestMessage: Action(fix or rebuild), Vulnerability, Image, BaseImageVersion(map to provide the base images' version if rebuild)

ResponseMessage: Success(bool), Vulnerability, Image, FixVersion



Concurrency Control and RPC

1. Goroutine for starting different RPCs to communicate with different image builders.

2. Thread-safe map with read and write mutex to ensure the consistency of intermediate results.

3. WaitGroup to coordinate running of different goroutines.

4. Channel for time out control

Reliable Delivery - At Least Once Semantics

Coordinator Side

1. Send RPCs to servers and wait

2. If there's timeout, resend the request.

3. Get response from server, parse the response and store the fix information in map (fixed images to the new versions). If get duplicate response, discard. If get an error response, resend the request

Server Side

1. Receive RPC requests and parse requests. If get duplicate requests, resend the response.

2. Fix/rebuild image based on request, and store the fix information locally. (So if get duplicate requests, directly resend the previous information)

3. Send the response back to the client

Future work

1. Vulnerability filter: filter for some vulnerabilities with specific requirements (severity, package related, etc), and for those vulnerabilities that have been fixed in other version, just need to rebuild instead of fix.

2. More efficient concurrent algorithm: goroutines do not need to block each other. More efficient delivery semantics, ex. exactly once

3. If fix/rebuild fails, we terminate the program immediately at present. We could have it running and continuing fixing other images that are not related with the failed image. Also, we can save the images' fix/rebuild information in disk for future use.

4. Now the coordinator manages the overall system working. We could share information to different servers and synchronize among them periodically. So that the servers could directly talk to each other without coordinator.

Thank you!