Everfix: Smart Vulnerability Remediation

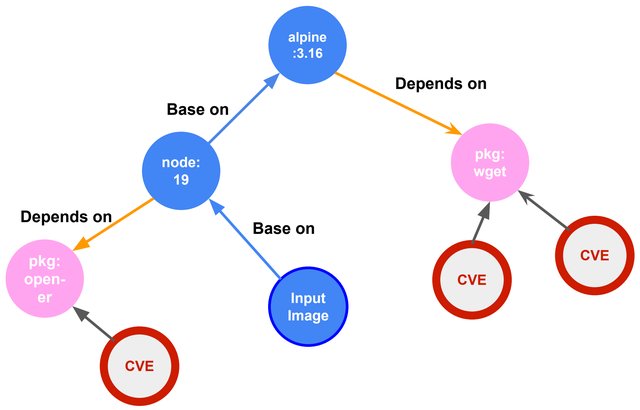
Team 7: Yinan An, Jingyi Huang, Xinzhe Jiang, Pingcheng Dong

Objective

Vulnerability management is a rather important security problem in modern software supply chains. An example of this problem would be the log4j vulnerability, which caused great damage to a lot of Java application. For container images, the “fix where discovered” strategy would be inefficient as they are built layer-by-layer.

In this project, we built a vulnerability remediation system to suggest smart fix workflow. That is, to find the provenance of vulnerabilities to fix and rebuild images depending on the vulnerable packages.

Remediation Example

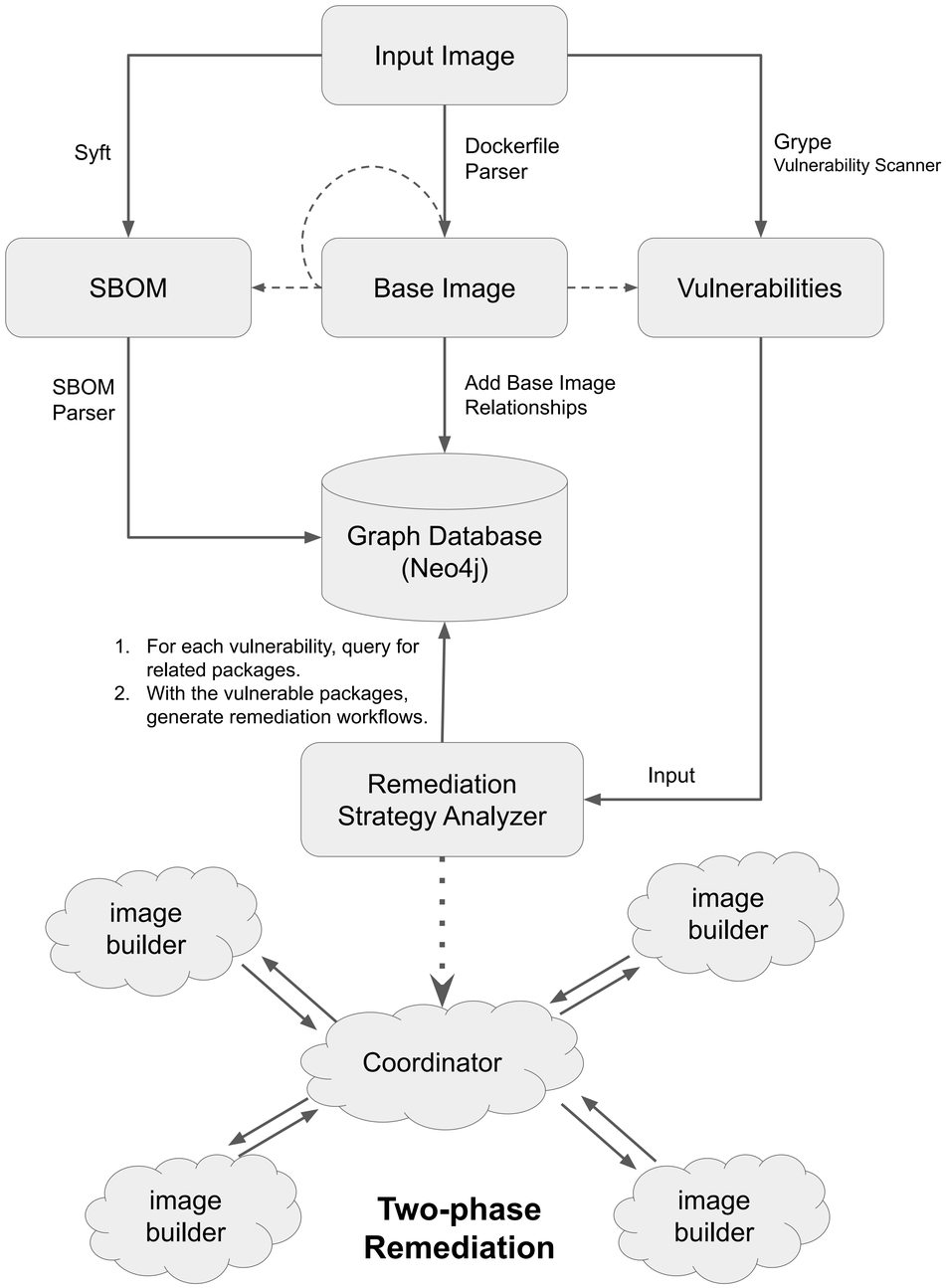


1. For vulnerable package wget, fix alpine (by upgrading wget version) then rebuild node and the input image.
2. For vulnerable package opener, fix node and rebuild the input image.

Further Discussion

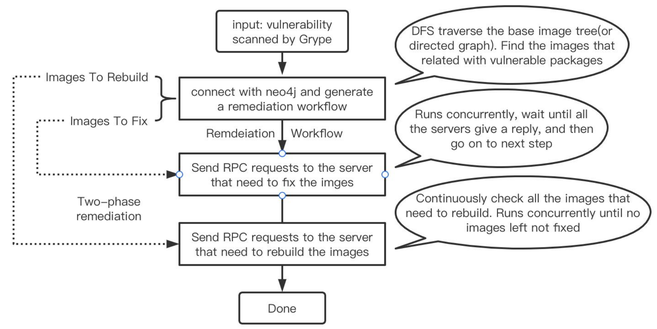
1. Vulnerability filter: filter for some vulnerabilities with some specific requirements (severity, package related, etc), and for those vulnerabilities that fix in other version, just need to rebuild instead of fix.
2. More efficient concurrent algorithm: goroutines do not need to block each other.
3. More efficient delivery semantics, ex. exactly once
4. 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.

Architecture



Implementation

1. Data structure:
a. Config
Initialized at the very beginning from the config files and is effective all the life cycle of the program. It contains ImageConfig (Image information that we want to remediate), RPCConfig (timeout and retry times), and AddressConfig (Destinations for RPCs)
b. Vulnerability
Name, PkgName, Severity, ConnectedImages (to store all the images that connected with the vulnerable package), RemediationSteps (to store the remediation steps that generated based on neo4j relations), FixStatus (a thread-safe map to store the fixed version of related images)
c. RemediationSteps and Step
RemediationSteps is a list of Step, with each Step stores the operation (fix or rebuild) and the images that this operation is on.
d. ImageTreeNode
ImageName, []*ImageTreeNode. The overall data structure stores all the base-image relationships for generating remediation strategy.
e. RequestMessage and ResponseMessage
The data structure of the RPC Message in 2-phase remediation.
RequestMessage: Action(fix or rebuild), Vulnerability, Image, BaseImageVersion(map to provide the base images' version if rebuild)
ResponseMessage: Success(bool), Vulnerability, Image, FixVersion
2. Remediation Workflow



3. Two-phase remediation
a. Concurrency Control:
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(threads).
b. Run in Each Phase
Get all the images that could be fixed or rebuilt; Send RPCs and wait for responses; Save responses and start another run.
c. Reliable Delivery
At-least once semantics: resend the request after timeout; discard duplicate requests and responses.