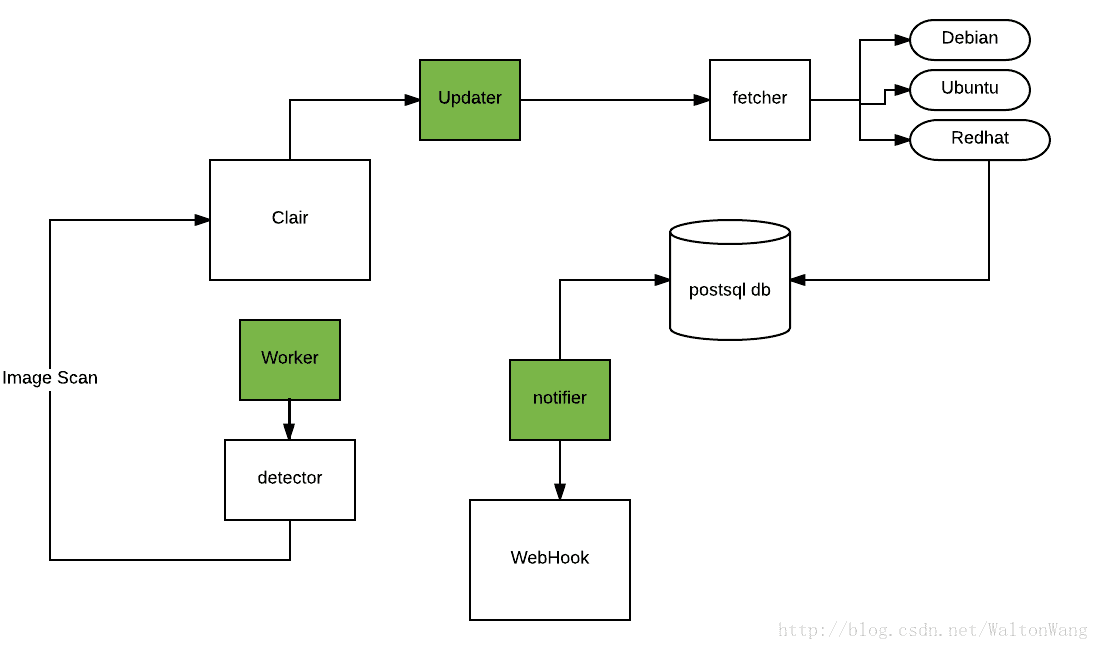
**Clair原理分析**

**Clair主要模块：**



* Fetcher（获取器）- 从公共源收集漏洞数据
* Detector（检测器）- 指出容器镜像中包含的Feature
* Notifier+WebHook（通知钩子）- 当新的漏洞被发现时或者已经存在的漏洞发生改变时通知用户/机器
* Databases（数据库）- 存储容器中各个层以及漏洞
* Worker（主进程） - 每个Post Layer都会启动一个worker进行Layer Detect

**Clair的名词解释：**

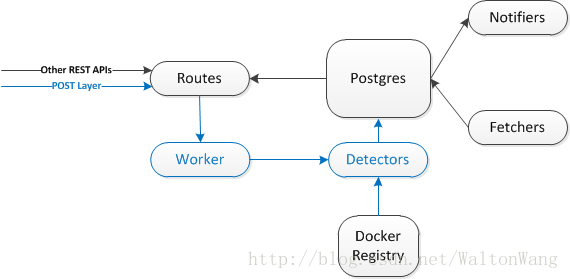
## Container

* Container - the execution of an image
* Image - a set of tarballs that contain the filesystem contents and run-time metadata of a container
* Layer - one of the tarballs used in the composition of an image, often expressed as a filesystem delta from another layer

## Specific to Clair

* Ancestry - the Clair-internal representation of an Image
* Feature - anything that when present in a filesystem could be an indication of a vulnerability (e.g. the presence of a file or an installed software package)
* Feature Namespace (featurens) - a context around features and vulnerabilities (e.g. an operating system or a programming language)
* Vulnerability Source (vulnsrc) - the component of Clair that tracks upstream vulnerability data and imports them into Clair's database
* Vulnerability Metadata Source (vulnmdsrc) - the component of Clair that tracks upstream vulnerability metadata and associates them with vulnerabilities in Clair's database

**代码流程：**



/cmd/clair/main.go：

**func main() {**

……

//加载配置文件

config, err := LoadConfig(\*flagConfigPath)

…

//配置updater和worker

configClairVersion(config)

//启动clair

Boot(config)

**}**

//从config文件获取参数，然后启动clair

**func Boot(config \*Config) {**

……

//连接database，默认配置pgsql

db, err := database.Open(config.Database)

……

//启动Notifier服务

st.Begin()

go clair.RunNotifier(config.Notifier, db, st)

//启动Clair的Rest API服务

go api.Run(config.API, db)

//启动Clair的健康检查端口

st.Begin()

go api.RunHealth(config.API, db, st)

//启动Clair的updater服务

st.Begin()

go clair.RunUpdater(config.Updater, db, st)

//监听终止信号

waitForSignals(syscall.SIGINT, syscall.SIGTERM)

log.Info("Received interruption, gracefully stopping ...")

st.Stop()

**}**

/api/v3/rpc.go：

//PostAncestry上传ancestry通过Clair的GRPC服务

func (s \*AncestryServer) **PostAncestry**(ctx context.Context, req \*pb.PostAncestryRequest) (\*pb.PostAncestryResponse, error) {

//获取ancestry名字

ancestryName := req.GetAncestryName()

……

//获取layers

layers := req.GetLayers()

……

//获取ancestry格式

ancestryFormat := req.GetFormat()

……

//获取ancestry的layers

ancestryLayers := []clair.LayerRequest{}

……

//调用worker.go的ProcessAncestry

err := clair.ProcessAncestry(s.Store, ancestryFormat, ancestryName, ancestryLayers)

……

}

/worker.go：

//ProcessAncestry扫描一个ancestry（本地没有则从网上下载）

**func ProcessAncestry(datastore database.Datastore, imageFormat, name string, layerRequest []LayerRequest) error {**

//调用isAncestryProcessed判断当前ancestry是否已被处理

if ok, err := isAncestryProcessed(datastore, name); ok && err == nil {

……

//调用processLayers处理layer

layers, err := processLayers(datastore, imageFormat, layerRequest)

……

//最后再继续调用processAncestry

return processAncestry(datastore, name, layers)

**}**

//processLayer处理一系列操作，请求layers，存储layers，最后返回一个扫描出features和namespaces的layer列表

**func processLayers(datastore database.Datastore, imageFormat string, requests []LayerRequest) ([]database.LayerWithContent, error) {**

……

//从数据库获取layer

layer, preq, err := getLayer(datastore, req)

……

//调用processRequests解析出所有layer里的features和namespaces

namespaces, features, partialRes, err := processRequests(imageFormat, toDetect)

// Store partial results.

if err := persistNamespaces(datastore, namespaces); err != nil {

return nil, err

}

if err := persistFeatures(datastore, features); err != nil {

return nil, err

}

for \_, res := range partialRes {

if err := persistPartialLayer(datastore, res); err != nil {

return nil, err

}

}

completeLayers := []database.LayerWithContent{}

for \_, req := range requests {

if partialLayer, ok := partialRes[req.Hash]; ok {

completeLayers = append(completeLayers, combineLayers(layers[req.Hash], partialLayer))

} else {

completeLayers = append(completeLayers, layers[req.Hash])

}

}

return completeLayers, nil

**}**

**//**processRequests接收layers，然后返回namespaces和features

**func processRequests(imageFormat string, toDetect []processRequest) ([]database.Namespace, []database.Feature, map[string]partialLayer, error) {**

……

//核心函数detectContent，解析layer

res.namespaces, res.features, res.err = detectContent(imageFormat, req.request.Hash, req.request.Path, req.request.Headers, req.notProcessedBy)

……

return namespaces, features, updates, nil

**}**

//detectContent下载一个layer然后对其进行扫描，解析处特征和操作系统

f**unc detectContent(imageFormat, name, path string, headers map[string]string, toProcess database.Processors) (namespaces []database.Namespace, featureVersions []database.Feature, err error) {**

……

**//调用Extract将指定的所有layers进行下载并解压缩（/ext/imagefmt/driver.go/Extract()->/pkg/tarutil/tarutil.go/ExtractFiles()->NewTarReadClose() ）**

files, err := imagefmt.Extract(imageFormat, path, headers, totalRequiredFiles)

……

**//调用featurens.Detect解析出镜像所对应的OS信息**

**（ /ext/featurens/driver.go/Detect()->/alpinerelease/ alpinerelease.go | aptsources/aptsources.go | lsbrelease/lsbrelease.go | osrelease/osrelease.go | redhatrelease/redhatrelease.go / Detect() ）**

namespaces, err = featurens.Detect(files, toProcess.Detectors)

……

//调用featurefmt.ListFeatures解析出镜像中所包含的所有软件信息

featureVersions, err = featurefmt.ListFeatures(files, toProcess.Listers)

……

return

**}**



alpinerelease：通过/etc/alpine-release检测OS是否为alpine，以及alpine的版本

aptsources：通过/etc/apt/sources.list检测OS是否为Debian或Ubuntu，以及对应的版本

lsbrelease：通过/etc/lsb-release查询OS的详细信息

osrelease：通过/etc/os-release和/usr/lib/os-release查询OS的详细信息

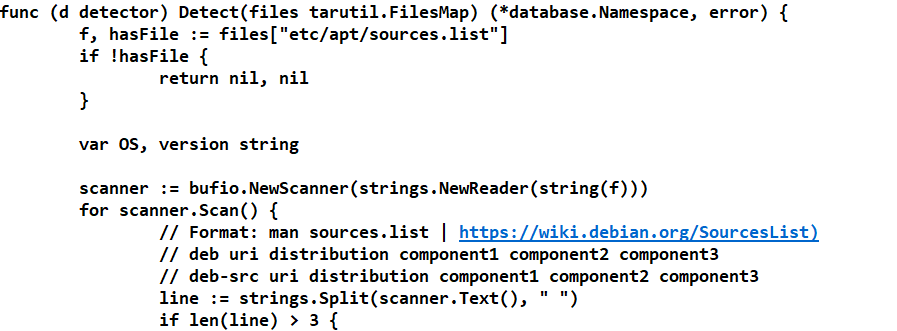
redhatrelease：查询redhat系列的OS的详细信息

/alpinerelease.go：

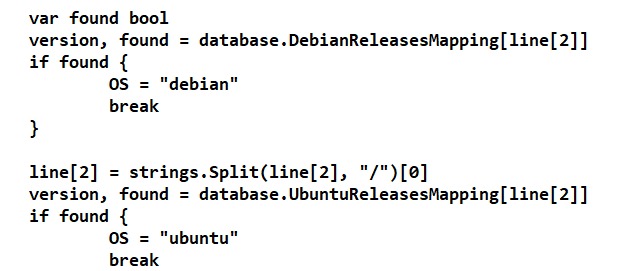


通过扫描/etc/alpine-release文件判断是否为alpine，然后从文件中读取alpine版本

/aptsources.go：

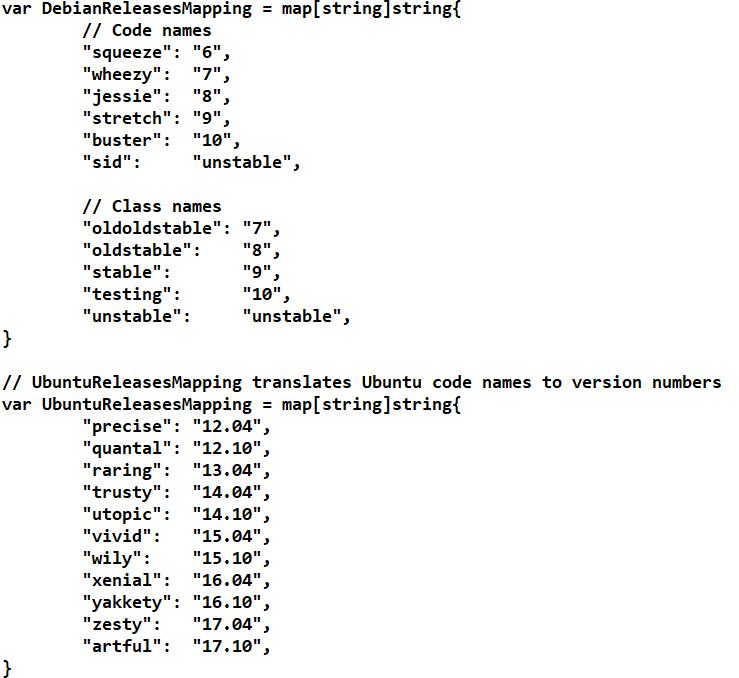


获取apt源文件内容



分别调用/database/namespace\_mapping.go的函数

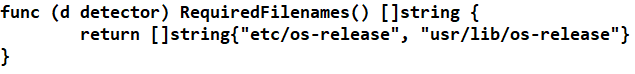
DebianReleasesMapping和UbuntuReleasesMapping进行识别



/lsbrelease.go：



/osrelease.go：



/redhatrelease.go：

