Malware Detection within Object Storage

Author: Matthew Battagel, Supervisor: Theodoros Spyridopoulos

Acknowledgments -

I would like to extend my sincere gratitude to my supervisor Theo, my colleague Harry, friends, family, and Lo for their unwavering support and encouragement during my project. Their combined expertise and guidance provided were critical in the shaping and execution of the project. I am truly grateful to all of them for their contributions.

Abstract

Lorem Ipsum

Contents

1. Introduction

Overview

The exponential growth of data generation has made data storage an increasingly important aspect for both individuals and organizations alike. Object storage has emerged as a promising solution due to its ability to store vast amounts of unstructured data in a cost-effective and scalable manner. Unlike traditional storage techniques, object storage stores data as objects with related metadata and unique identifiers, allowing for efficient and cheap storage within buckets.

One of the most widely used object storage platforms is Amazon S3, which provides a highly scalable and reliable solution for storing data. However, an open-source alternative called MinIO has emerged as a promising contender, providing similar features to Amazon S3 while giving customers greater control over their data. MinIO is written in Go and is available for free under the Apache License 3.0 or, for commercial and enterprise purposes, at a reduced cost compared to Amazon S3 (**?**). MinIO offers a wide range of features, including high performance, data replication, encryption and erasure coding (MinIO, 2023b). Most importantly, MinIO is designed to scale out horizontally to ensure that it can handle the demands of large-scale applications.

Scalability is made simple by allowing multiple types of hardware platforms to work together in separate nodes each with their own compute and storage. This is extremely attractive for customers who want to utilises their existing hardware without being tied down to a specific provider. This also applies for customers looking to migrate their data from Amazon S3 to cheaper solution without compromising on the high performance, reliability and scalability of the S3 platform.

While MinIO is a great alternative to Amazon S3, it does not offer any form of malware detection integration. This could put customers off from choosing MinIO as a viable platform

to migrate to from Amazon S3 or leave existing users data vulnerable to malware attacks. This project aims to address this issue by integrating a malware detection system into MinIO. An important goal for the is to negatively impact the scalability or performance as little as possible so that MinIO is still an effective alternative to Amazon S3.

Motivation

Due to the high amount of unstructured data expected to be both written and read to the object store, there are increased risk of encountering malicious files. Therefore malware detection within object storage is crucial in modern cloud storage scenarios. Most popular off-the-shelf object storage platforms, such as AWS, already have integrated third-party antivirus software, such as ClamAV and Sophos (Srinivasan *et al.*, 2022), to mitigate security risks. MinIO on the other hand is vulnerable to malware attacks as it currently does not have any native antivirus integration. This forces customers who require complete virus protection to either not use MinIO or to use potentially costly third-party software. As antivirus scanning is inherently resource intensive, if the software is integrated incorrectly, it could reduce the ability for the storage solution to scale horizontally which negates one of the major benefits of object storage. The purpose of this project is to implement malware detection within MinIO while being mindful to not impact the scalability or performance of the platform.

Project Aims

From a personal perspective, by completing this project

2. Background

Amazon S3 Malware Detection

As MinIO's largest competitor, this project draws a lot of inspiration from Amazon S3s integrated malware detection blog page (Srinivasan *et al.*, 2022). The blog explains Amazons current approach for managing malware detection within their service. Amazon S3 uses a combination of ClamAV and Sophos as their third-party scanning engines due to their out-of-the-box nature. Amazon then gives you the option to use either of these engines or both. The blog goes on to describe the three main interaction mechanisms that Amazon S3 uses to flag files for scanning. Firstly, an API endpoint would be provided to handle all uploads. This forms a queue of uploads which are then scanned before entering the bucket. Next, event-driven scanning is used keep track of all regular file uploads. The antivirus will then scan each file after they have been written to the bucket. Finally, retro-driven scanning is used to scan all existing files within the bucket. The user then has the flexibility to define what types of files should be scanned including defining time windows. This blog has given some useful methodologies of how to keeping track of both incoming and previously scanned files. Creating a system that can match these methods is important for offering a matching level of scalability and security within MinIO.

3. Specification

The specification for this project is to help guide the project to fulfill the aims set out in the previous section. The specification is broken down into three main sections; functional requirements, non-functional requirements and constraints.

Functional Requirements

Functional requirements are used to define how the solution must work for the project to be considered a success. This project aims to supply an end-to-end solution for detecting malware within the MinIO object storage platform. This goal can be separated into a list of functional requirements:

- Provide a way of detecting the latest uploads to the object store.
- Record the results of the malware detection within the object store.
- Provision for future expansion and ongoing maintenance.
- Have a high level of customisability to allow for different use cases.
- Allow for efficient and transparent debugging in the event of failure.
- Provide the ability to measure various metrics.
- Scale alongside MinIO to ensure that it does not bottleneck the object store at high loads.

Non-Functional Requirements

Non-functional requirements are used to define the quality of the solution that is required. This project aims to be production ready and therefore the solution must be held to a high standard. These high standards come in the form of ambitious targets in which the solution will have to satisfy in order to be considered production-ready. The non-functional requirements are split into five categories with metrics to measure their success.

- Speed - The solution must be able to keep up with the rate of uploads made to MinIO. This can be measured by comparing the time difference between uploading a object to MinIO and the object being scanned and tagged.
- Availability - Over a long period of time the solution must be able to handle all requests. This can be measured by comparing the number of requests made to the number of requests completed over a large time frame.
- Capacity - The solution must be able to handle the maximum number of simultaneous requests that MinIO can handle. This can be measured by monitoring the amount of cache used by the solution under load.
- Reliability - 100% of the files uploaded to MinIO must go through the scanning process. The recorded metrics can be used to compare MinIO uploads with the number of objects scanned. It is worth noting that checking the clean and infected results add to the total sum of scanned objects
- Usability - Future additions, maintenance and debugging must be as simple as possible. This requirement is more subjective and therefore explanation of how I have achieved this will be discussed in the implementation section.

Constraints

The constraints are the limitations that the solution must adhere to. The main constraint of the project is the strict time limit given to the project. There are a total of 12 weeks to achieve a production ready product which will greatly limit the scope of the project. This means accurately prioritising the features that are most important to the project while also balancing the time spent to implement them. The second constraint is the limited resources available to the project.

Another constraint of the project is that all the external software used must be open source / available for commercial use under license or fee. This is to ensure that the project is legally viable if the solution was to be used commercially.

The final constraint is that my own knowledge and experience will increase the average time taken to implement milestones. This is due to the fact that I will need to both include time to learn each new technology and also allot excess time if I incorrectly size a task.

4. Architecture

Choosing the correct architecture for the project is critical for ensuring that the solution is scalable, performant and maintainable. Given the specification above, various potential architectures can be created and evaluated based my own thoughts and from reading the background material. The best candidate design will then be chosen based on which candidate design satisfies the most requirement with as little compromises as possible. Thought will also be given to which architecture fits within the constraints of the project.

Candidate Design 1 - Post-Write

The first candidate design makes use of the performance benefits of MinIO by allowing puts to be initially written to the bucket without being scanned. The design then uses a event queue compatible with MinIO to keep track of all the files that have been uploaded. The queue is then used to trigger a scan of the file once an antivirus is available. The candidate design is shown in figure 1.

This design has many benefits over other potential implementations. Firstly, it uses the storage provided by MinIO to store all incoming files without having to manage a separate storage solution. This removes a lot of complexity from the solution by not having to account for a number of failure conditions that could occur with a high availability, production ready storage solution. For example, the solution would not be responsible for handling partial writes, loss of data, or data corruption. Removing this responsibility allows the solution to focus on the core functionality of the project, the scanning of files, which is essential for keeping the project within the time constraints.

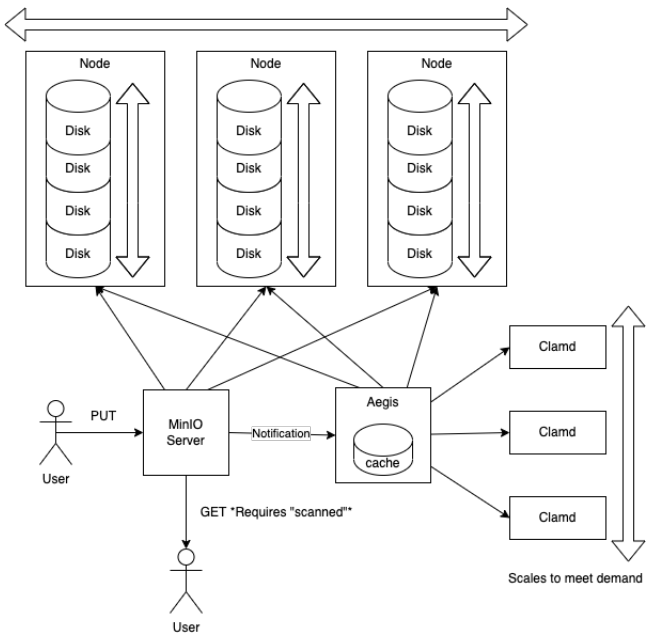

Figure 1: Post-Write Architecture

Secondly, the design also makes use of the integrated event queue provided by MinIO. This again removes responsibility from the solution by differing the scalability and reliability requirements of an event queue to MinIO.

Lastly, having Aegis dispatch the files to a scalable number of antivirus scanners allows the solution to scale to meet the demands of the system. This meets a key requirement as the solution is expected to have the capacity for a large number of operations. This method does require the use of a load balancer to effectively distribute the load across the available antivirus scanners.

The candidate design also has a number of drawbacks. Firstly, the design still requires a small about of cache to temporarily store the object when it is being dispatched to the antivirus. Provisioning of this cache has to be large enough to handle the largest file possible to be uploaded to the object store. In reality, this cache would be provisioned even larger to allow for the temporary storage of multiple objects while multiple scans are being performed asynchronously. In addition, the cache needs to be large enough to ensure that the system does not become overwhelmed by the number of objects being scanned as the system scales. This is a minor issue as store capacity is cheap and the provisioning of the cache easy to scale up. Additionally, a higher priority can be given to scaling up and out antivirus scanners to ensure that the smallest number of files are being cached, while bring scanned, at any point.

The second drawback is that, for each event, Aegis makes a get request for the object to be scanned. This effectively doubles the number of requests made to the object store. This also means that Aegis must have the ability to get any file expected to be scanned and therefore must have access to the whole storage network. The impact of this drawback is mitigated as the solution is expected to be deployed on the same network as the object store which should reduce the latency of each request made by Aegis. However, this still leaves MinIO to handle twice as many requests with the performance loss being noticed mainly on more distributed storage topologies.

Thirdly, the candidate design only allows for a single Aegis instance to dispatch all incoming objects to available scanners. This is a potential bottleneck for the system as this instance could become overwhelmed by the number of requests it is receiving. This is a minor issue as the dispatching of objects to scanners is not as performance intensive as other areas of the solution, such as the actual scanning, and therefore it is not expected to be a major bottleneck.

Lastly, any object uploaded to the store will have a certain period of time where it remains unchecked. In this time, the user could potentially download an unscanned object or the object could cause harm to the store before it is detected. Although the handling of infected objects is out of scope, in an actual implementation of the solution, the user could be made unable to download unscanned objects until they have been scanned.

Candidate Design 2 - Upload Queue

This candidate design created a wrapper around MinIO that the user interacts with instead of MinIO. This means that all puts go through Aegis before being uploaded to the object store. The candidate design is shown in figure 2.

The main benefit of this design is that the user interacts only with Aegis when uploading files. This means that all incoming files can be stored within a temporary storage before ever entering the object store. This offers the best protection against malicious files as the user cannot ever download an unscanned or infected file as it is never uploaded to the object store. Infected files can then either be deleted or moved to a separate quarantine store for analysis.

This candidate designs main advantage also comes with a major drawback. This design requires Aegis to handle the full throughput of all the puts to the system. Aegis then has the full responsibility of being available to all puts and, in a failure scenario, to handle the recovery of the system. Additionally, the cache provisioned must be large enough to handle the largest files at maximum throughput with extra room for unexpected delays. This negatively affects the scope of the project by requiring the solution to prioritise features that are already covered by MinIO.

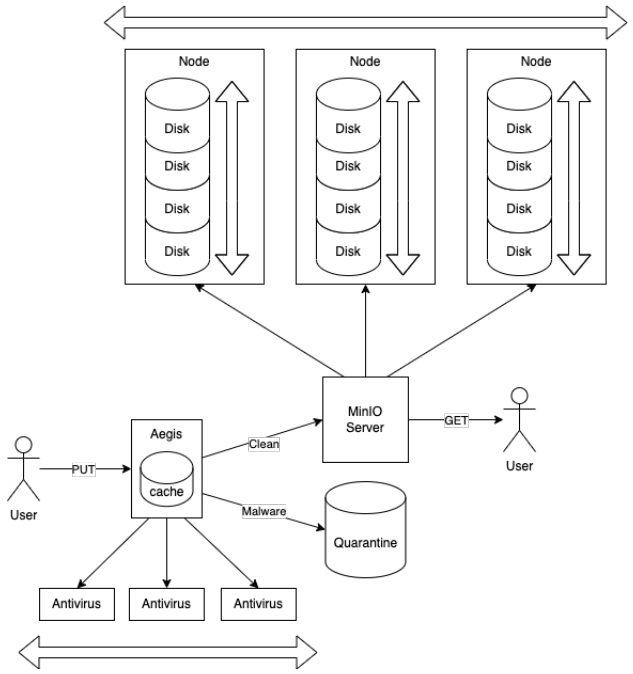

Figure 2: Upload Queue Architecture

Because MinIO is dependent on Aegis to handle the puts, MinIO must wait to be passed incoming objects sequentially after Aegis has finished processing the previous object. This removes the potential for aggregate performance where

Candidate Design 3 - Write Interception

Candidate Design three is very similar to the second candidate design, however, instead of wrapping outside the MinIO service, it intercepts the writes from the client before objects are written to the object store. With this interception, Aegis can scan the object and decide whether to allow the object to be written to the store or to quarantine the object. The candidate design is shown in figure 3.

This candidate design has similar benefits as the second candidate design. It offers the most protection against malicious files by never allowing either unscanned or infected objects to be stored in the object store. However, it also has similar drawbacks. This is because Aegis is still in sequence with MinIO meaning that for optimal throughput, Aegis would need to match the performance of MinIO.

Similar to the upload queue candidate design, this design also requires Aegis to have a large cache to handle the largest files at maximum throughput. This cache must also be large enough to handle the

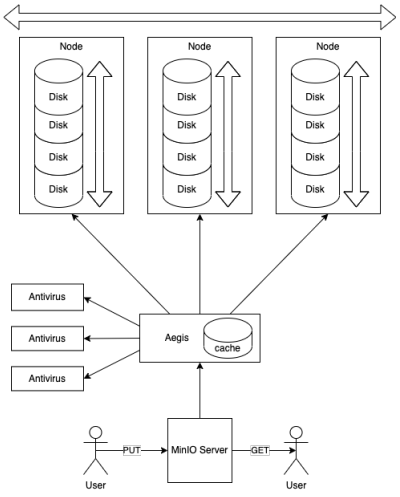

Figure 3: Write Interception Architecture

number of objects being put by MinIO into the store. This issue cannot be mitigated without the risk of compromising performance at increased loads.

However, this design does have an advantage over the second candidate design as there is less responsibility placed on Aegis to be as failure tolerant. MinIO is still directly responsible for accepting objects into the store and therefore is still responsible for the recovery of the system in a failure scenario. This allows the scope to focus on more related features to malware scanning.

Candidate Design 4 - Per Node

The final candidate design distributes Aegis onto each node in the object store. This means that each node has a local instance of Aegis that is responsible for scanning objects before they are written to the store. The candidate design is shown in figure 4.

This candidate design makes use of the distributed nature of MinIO to match the demand when scaling out the system. As more nodes are added, more Aegis instances are added to handle the increased scanning demand. This removes the need for having a cache repository as Aegis already has access to the files that need scanning. By removing this single point of failure, in theory, the system only relies on the antivirus pod to be able to scale out on its own.

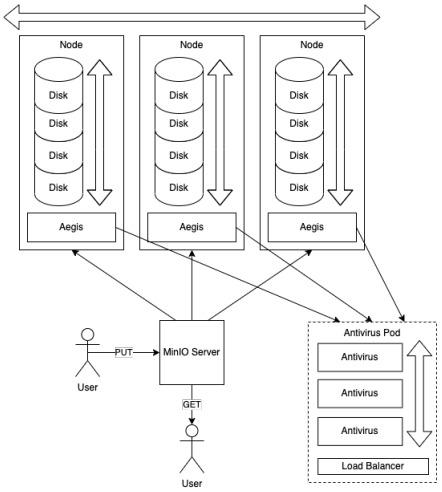

Figure 4: Antivirus per Node Architecture

Independent scaling of the antivirus pod allow for efficient usage of available hardware. A simple load based auto-scaler can be used to scale the number of pods based on the current load. This allows for the system to flexible scale with the demand of the system and to reduce usage of valuable resources, such as power. There is also the opportunity to use intelligent scaling techniques to predict the load on the system and prematurely scale the system to meet the demand. For example, to scale the number of pods depending on the time of day or the day of the week.

The major drawback of this candidate design is that it replies on the ability to scan whole files by only using data on a single node. In actual implementations, MinIO makes use of erasure coding to add increased redundancy to the store (MinIO Erasure Coding, 2023). Erasure coding splits objects into multiple parts known as blocks, and then calculates corresponding parity blocks. These data and parity blocks are then distributed among all nodes in the system allowing for on-the-fly data recovery even with the loss of multiple drives or nodes . This means that the Aegis instance on each node only has access to the part available on their node and therefore will not be able to reconstruct the whole file for scanning. This makes this candidate design unsuitable for MinIO as it erasure coding is one of its key features.

Selected Candidate Design

Given the above evaluations of each candidate design, design one best meets the requirements and constraints of the project. It makes the most use of the existing features that MinIO provides in order to handle failure scenarios and to scale out. This also means that this design has less critical responsibility and will better fit the scope constraints allowing for more time to be spent on supplementary features, such as testing, logging, and metric collection. Because of this, the produced solution will be closer to production-ready than the other candidate designs.

This candidate design keeps the user in control by giving them the ability to store unscanned files / known malware without wasting resources on a scan. Protection can be added per bucket therefore a user could have a known malware bucket and a clean bucket within the same object store. This allows for the system to be more flexible and to be able to handle more use cases. Candidate Designs two and three would not be as able to handle this use case as they both scan all objects before they are written to the store.

The size of the cache required is smaller than all other candidate designs as it only needs to store the objects actively being scanned. This is in opposition to upload queue and write interception candidate designs as they have to be prepared to handle the full demand placed on the store. This makes candidate design one the most lightweight of all the candidate designs which should lead to a smaller resource footprint.

Overall, implementing candidate design one is the best option to create a lightweight yet secure and scalable solution.

5. Implementation

Service Selection and Creation

Before I started to code the solution, I needed to research, create, configure and most importantly understand each external service that I would be using. Up to this point I know what types of services I would need to create, but I did not know what specific services I would be using. This section will go through the process of how I chose each service and then initially configured them to form a bare-bones proof of concept.

Object Store - MinIO

The object store is the only service that I did not need to research or compare as it is already the subject of this project. However; I did need to create and configure a local instance of MinIO for development. MinIO itself is available from many sources, including Docker, Homebrew, and the MinIO website. I chose to use Homebrew, a MacOS package manager, as it is the easiest to install and update. I then used the MinIO documentation to create a local instance of MinIO where I can access the web client on http://localhost:9000. From this I could create buckets, upload objects to the store and get more familiar with MinIO's features.

MinIO has integrated the ability to send notifications to event queues depending on what operation is performed on the store. This makes it quick and easy to set up a locally running instance of an event queue to read messages sent by MinIO. MinIO offers wide support for many different event queues, such as Kafka, Webhook, Redis, PostgreSQL, and many more.

Event Queue

Kafka is a popular and well-supported event queue that is used in many different industries. It offers many features that make it a good choice for this project, such as high-throughput, low latency and open-source. Kafka is also available from both Docker and Homebrew make it easy to install and run locally. Most modern event queues offer similar high throughput and low latency, but Kafka is lighter in resource usage than other queues, such as RabbitMQ (Levy, 2022). This is beneficial as it will be easier to implement and not over use resources when it has a simple use case.

Kafka has a dependency on Zookeeper, which is a distributed coordination service. Zookeeper must be installed as a separate service but is available from the same sources as Kafka. Kafka can be run in Kafka Raft mode (KRaft) which will eventually replace Zookeeper, but as of writing KRaft has not been fully adopted yet (M, 2022).

With Kafka and Zookeeper setup, I use Kafka's command line interface (CLI) to start the service and create a topic. I then use the MinIO documentation to configure MinIO to send all put notifications to the Kafka topic. I can then use the Kafka CLI to read messages from the topic and see the messages sent by MinIO. This proves that the event queue is working and that MinIO is sending messages to it whenever I perform a put operation. An example Kafka message is available in the appendix in listing 1.

Antivirus

The antivirus we choose has to meet a list of requirements for it to be suitable for use in this project. Firstly, it must be able to be scaled with the solution in order to keep up with the demand placed on the system. Secondly, it must have a CLI that our program can interact with in order to scan files. Finally, it must be free or as cheap as possible to make it viable for commercial use. This narrows down the available options to a few contenders.

Sophos is a popular antivirus that is used by many businesses and is available for free for personal use, however; is it paid

ClamAV however is completely free and open-source. It comes with a scalable and multi-threaded daemon which can be accessed via CLI for high-performance and on-demand file scanning. (ClamAV, 2023). It is capable of scanning many different file types, including archives and mail files. Build-in is freshclam, a tool for automatically updating the virus database definitions. The virus database itself is also open-source and is updated regularly by the open source community. ClamAV has a docker image and is available from Homebrew.

I have chosen to use ClamAV as it is totally free for commercial use, open-source and has a CLI that can be used to scan files. It is also very well documented and has a large community of users. Sophos

ClamAV comes with a daemon, clamd, that can be run in the background and can be accessed via a CLI using clamdscan, the clamd client. A configuration file is needed to point clamdscan to the IP address that the daemon is running. In this case it is running locally on port 3310. Performing the clamdscan command and providing a file will scan the file and return the result. An example of this is available in the appendix in listing 2.

Metric Collection

As the project is expected to be as production-ready as possible, a system for collecting metrics is needed. This will allow the system to monitor its activity for easier maintenance and debugging. A few different options are available for metric collection, such as Prometheus, InfluxDB, and Graphite. Each of these

From this comparison, I chose to use Prometheus as it is open-source, uses a pull model with the option for a push gateway and it does not require a distributed system to run, unlike InfluxDB. Prometheus is available from both Docker and Homebrew.

Prometheus is currently unusable as I am not sending it any metrics to collect. I can still launch the Prometheus server and access the web interface on port 9090. In the meantime I can use the Prometheus documentation to create a configuration file defining the port and endpoint I expect to be exposing metrics to, in this case port 2112 and /metrics.

s

Audit Log Store

Production-ready software should have a way to store logs for auditing and analysis purposes. This is a relatively simple requirement as a central database can be used due to the expected low amount of data needing to be written. The purpose of the audit log is to store information about the scans performed by the system with information such as, time and result of the scan as well as the antivirus used. This gives the system a way to look back at previous scans which could help with debugging and maintenance.

There are many different options for a database to store the audit logs, such as PostgreSQL, MySQL, MongoDB, and many more. PostgreSQL remains the most versatile out of these options as it is a relational database and can be used for many different purposes. As the use case is simple, keeping to a single database service is the best option for maintainability and usability. PostgreSQL is also available from both Docker and Homebrew.

Like Prometheus, PostgreSQL is currently unusable as I am not sending it any data to store. However, I can still launch the PostgreSQL server and access the database via PostgreSQL shell prompt (psql). I can create a database and a user for the database to use. This database can then be exposed to port 5432 on the localhost ready for Aegis to use.

Data Visualisation

Grafana

Compare

Docker

Aegis Design and Creation

Now all of the dependencies have been downloaded, initialised and configured, Aegis can now be coded to interact with them.

The most common language to use for micro-service based projects is GoLang. Go is a compiled language that is statically typed and high concurrency support. This makes it a perfect fit for this project to ensure that the system is as performant as possible. MinIO is also written in Go should allow for easier integration. Go has certain standards and guidelines for clean architecture that should be followed to ensure high usability and efficiency.

Project Structure

In Go you should the separate the code into different packages, where each package represents a distinct function of the system. This allows for easier maintenance and debugging in the future. These packages are then grouped into different directories depending on their intended scope and function. There are three main directories that are used in GoLang projects,

cmd, pkg and internal. The cmd directory is used to store the main.go file which manages the workflow and is the entry point of the program. The pkg directory is used to store all of the packages that are intended to be used by external applications, in this case our other services. The internal directory is used to store all of the packages that are intended to be used by the application itself. This is where the packages in the pkg directory will be consumed to perform the Aegis' core actions as follows:

- Listen for PUT events on the event queue
- Read the message from the event queue and extract the bucket and object path
- GET the object from the object store and store in cache
- Initiate a scan on the object using the antivirus software
- Collect the result of the scan and add tags to the object
- Collect metrics throughout the process
- Store the result of the scan in the audit log
- Expose metrics to Prometheus

From these requirements, the internal design of Aegis can be planned. We can visualise this plan using a class diagram. It is worth noting that Go itself does not have classed but instead uses structs which can be used to achieve the same effect. This plan is shown in figure 5.

From this diagram a file structure can be created inline with Go standards. Go comes with a CLI tool used to initialise a new go module, which is a collection of packages that are intended to be used together. This tool will create a go mod file which is used to define the module and its dependencies. This tool will also create a go sum file which is used to store the hashes of the dependencies to ensure that the same version is used across all environments. Now go commands can be used to install go dependencies and then download them to a local vendor folder for use in building. Building the project is also done with the go CLI. All binaries are stored within the build folder. The initial project structure is shown by the figure 6.

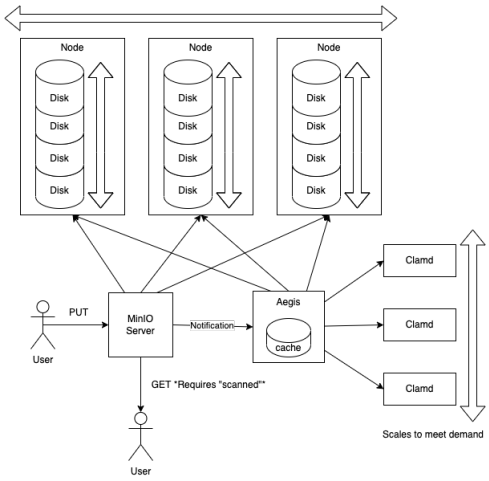

Figure 5: Plan of Aegis' Internal Packages

Version Control

I decided that due to the size of this project, that version control would be necessary to ensure that the project is maintainable and that there is always a backup available. I chose to use Git in combination with GitHub as I am very familiar with the platform and it is free to use. I used the Git CLI to initialise a new repository within the root directory of the go module. In addition, I created a git ignore file to ensure that the vendor and build folders are not uploaded.

In order to retain usability, throughout the project I will be committing to the remote GitHub repository

```
aegis/
├── cmd/
│   └── aegis/
│       └── main.go
├── pkg/
│   ├── clamav/
│   │   └── clamav.go
│   ├── config/
│   │   └── config.go
│   ├── kafka/
│   │   └── kafka.go
│   ├── logger/
│   │   └── logger.go
│   ├── minio/
│   │   └── minio.go
│   ├── postgresql/
│   │   └── postgresql.go
│   └── prometheus/
│       └── prometheus.go
└── internal/
    ├── auditlog/
    │   └── auditlog.go
    ├── dispatcher/
    │   └── dispatcher.go
    ├── events/
    │   └── events.go
    ├── object/
    │   └── object.go
    ├── objectstore/
    │   └── objectstore.go
```

after each significant change to the project while including relevant commit messages. This will make it easier to track changes for debugging and maintenance purposes.

Structured Logging

Structured logging is a method of logging that allows for easier parsing of the log messages by adding structure to the message. This is done by adding key value pairs containing relevant information to the log message. This enables for easier filtering and searching of the logs. This is especially useful when using a log aggregation tool such as Humio.

Go has many external modules that can handle this type of logging including Zap by Uber. Zap is a very fast and efficient logging library that can be configured to change the level of logging output, such as for info or debug useful for production or development respectively (uber go, 2023). Zap also has the ability to change between structured and unstructured logging depending on the use case. Structured logging comes with the drawback of being less human readable than unstructured logging and therefore Zap offers the ability to change between the two (uber go, 2023).

As Zap is an external module, it must be added as a dependency and all code contained in the pkg folder. A package called logger is created to encapsulate all logging functionality. This package contains two go files, one for interacting with the Zap and one for defining the structure of the log commands inside of an interface. In Go, it is idiomatic to keep the interface as close to the implementation as possible. Keeping the interface in the same package as the implementation allows for easier mocking of the package when testing. This package goes against this by containing the interface within the repository because multiple packages will need to use the same interface when interacting with them. This reduces code duplication as we don't have to define the interface in each package that uses it, which in this case would be every package. The code is shown in figure **??** and the interface is shown in figure **??**.

Configuration

As Aegis is a micro-service, it is important that it is configurable to allow for easy deployment to different environments. This is done by using a configuration file in tandem with Viper, a Go module that can read in the configuration file.

This configuration file contains all of the values that are likely to change between environments. This includes values such as the endpoints, ports and credentials of external services, logging options and database names. This configuration file will be a dotenv file which is a key value store of environment variables. This is beneficial as Viper has the ability automatically override the config file with environment variables if the same key is found. This allows for easier configuration of the application when it is built locally or deployed in a kubernetes cluster. This gives the user the ability to tailor the deployment to their needs or existing implementation.

Much like the logger package, the configuration package contains two go files. A repository file that defines the interface and a config file that contains the Viper configuration.

Adding both the configuration and structured logging as the first packages reduces the need for refactoring in the future. No hard coded values are needed for initial development as all values can be stored in the configuration file from the start.

Testing

Makefile

A Makefile is a file that contains a set of instructions that can be run from the command line. These instructions are used to automate processes such as building, testing and deploying. Through the project, longer workflows will be automated and put into the Makefile to reduce the amount of time spent on running commands. Makefiles also simplify the usage of program by users by encapsulating complex commands into an explicit action the user can understand. Makefile commands can also be used by the Dockerfile when building the application in a container.

External Package Integration

With the dependency services still running, the next step is to integrate them into the Aegis application using Go. This is done by creating a new package for each of the services within the pkg folder. Each package will contain a go file for each of the services that will later be consumed by Aegis' internal workflow.

One significant advantage of separating external services from internal implementation is that it allows for a higher level of abstraction from the services used. This abstraction provides flexibility in the use of multiple external services that fulfill the same purpose, such as multiple antivirus scanners. The creation of external packages allows for the use of a single internal implementation for all of these services. This reduces the need for extensive changes in the future, making the application more maintainable and scalable.

MinIO

The initial external service to be incorporated is MinIO. The MinIO package manages interactions with the MinIO service. For this project, the required operations include getting, putting, and removing objects, as well as getting and putting tags. MinIO provides a Go Software Development Kit (SDK) that already supports these operations in Go (MinIO, 2023a). The SDK simplifies the complexities of making requests and offers straightforward methods for operations such as putting and getting, as well as accessing type definitions like tags.

Once the MinIO SDK is imported, a new MinIO client object is generated to communicate with the MinIO service. The `CreateMinio` function accepts essential connection parameters, such as a context, an endpoint, access and secret keys, and an SSL usage flag, and initializes the MinIO client using the `minio.New` function. This client object is then incorporated into a custom `Minio` struct, along with a logger. Various methods are implemented for the Minio struct to execute different object storage operations:

- `GetObject`: Retrieves an object from a specified bucket and returns its data as a byte slice.

- `PutObject`: Takes in a byte stream and uploads it to a specified bucket with a specified object name.
- `RemoveObject`: Removes a specified object from a specified bucket.
- `GetObjectTagging`: Fetches the tags associated with an object and returns them as a map of key-value pairs.
- `PutObjectTagging`: Replaces the existing tags of an object with a new set of tags provided as a map of key-value pairs.
- `AddObjectTagging`: Adds new tags to an object by first fetching the existing tags, updating them with the new key-value pairs, and then setting the updated tags back to the object. Necessary for not overriding existing object tags that may exist.

All of these methods take in a context which is used in the shutdown process to close the connection to the MinIO service. This context is passed in during the creation of the MinIO struct so that all methods have access.

Kafka

The next external service to be integrated is Kafka. The Kafka package will handle the interactions with the Kafka service. For this project, the operations needed are to consume messages from a specified topic. Kafka provides a Go library, *kafka-go*, which simplifies the consumption of messages in a Go application (kaf, 2022).

The package imports necessary dependencies and creates a custom `KafkaConsumer` struct, which embeds a `kafka.Reader` object and a logger. The `CreateKafkaConsumer` function initializes a new `KafkaConsumer` instance by taking connection parameters such as the list of brokers, the topic to be consumed, a group ID, and a maximum number of bytes per message.

- `ReadMessage`: Uses the Kafka library to halt until a message is received from the specified topic. It then decodes it using the `decodeMessage` function, and returns the bucket name and object key.
- `decodeMessage`: Decodes a Kafka message by unmarshalling its JSON payload and extracting the bucket name and object key. If the message event is *s3:ObjectCreated:PutTagging*, it returns empty strings, as this event does not require processing.

During the shutdown process, the `ReadMessage` function is halted by closing the context passed in. This closes the connection to the Kafka service and therefore stops the reading of any new Kafka messages, leaving unprocessed messages in the event queue.

ClamAV

ClamAV is another external service to be integrated into the project. The primary operation needed for this project is scanning a file and returning the scan results. The ClamAV daemon can be interacted with through the command-line interface (CLI) using the `clamdscan` command (**?**).

Initially, a `ClamAVScanner` struct is created, embedding a logger. The `CreateClamAV` function initializes a new `ClamAVScanner` instance. Methods for the `ClamAVScanner` struct are implemented to perform various file scanning operations:

- `ScanFile`: Accepts a file path as an argument and scans the file using the built-in Go exec library to run the `clamdscan` command with the `--config` flag set to use a custom configuration file located at `clamav.conf`. The exec library enables the execution of external commands, providing the ability to interact with the ClamAV antivirus daemon.

A process attribute is added, which starts the process in a different process group than the main execution. This approach aids in achieving a graceful shutdown later on, as calling a system interrupt terminates the entire process group (McKusick *et al.*, 2014). As a result, `ScanFile` can continue executing after the shutdown, ensuring that no scans are interrupted. The method returns false if the file is clean, true if infected, and the type of malware detected. If any errors occur during the execution, it returns true (infected) along with an error message, ensuring that the worst-case scenario is assumed when it comes to security.

- `findVirusType`: Takes the output from the `clamdscan` command, extracts the virus type using regular expressions, and returns it as a string.
- `GetName`: Returns the name of the antivirus engine, in this case, ”clamav”. The name must be accessed through a method, as ClamAV will implement an interface that does not have access to any attributes.

The internal scanner package can now use the ClamAV package as one of the antivirus engines to scan files it receives.

Prometheus

The Prometheus package is in charge of creating and managing an HTTP server that exports metrics from Aegis. It does this by providing a plaintext response containing the metrics in the Prometheus exposition format. The exposed endpoint is then used by the Prometheus server to collect metrics from Aegis.

- `CreatePrometheusServer`: Initializes a new Prometheus exporter by setting up a new HTTP server with the specified endpoint and path. The Prometheus handler, provided by the `promhttp.Handler()` function from the Prometheus Go client library, is linked to the given path, which serves the plaintext. Read and write timeouts for the HTTP server are established using constants since these values are not expected to be configurable.
- `Start`: Initiates the Prometheus server by calling the `ListenAndServe()` method on the HTTP server. An example of the generated plaintext output can be found in the appendix at listing 3.
- `Stop`: Handles the graceful shutdown of the Prometheus server by invoking the `Shutdown` method to close the HTTP server.

PostgreSQL

The package imports necessary dependencies and creates a custom `PostgresqlDB` struct, which embeds a `pgxpool.Pool` object and a logger. The `CreatePostgresqlDB` function initializes a new `PostgresqlDB` instance by taking connection parameters such as the user, password, endpoint, and database name. It also returns a `CloseFunc` function to close the connection when needed.

- `CreatePostgresqlDB`: Is responsible for establishing a connection to the PostgreSQL database and returning a `PostgresqlDB` instance. The function takes in connection parameters such as the user, password, endpoint and database name. Additionally, it returns a `CloseFunc` function to facilitate a graceful shutdown of the connection pool when necessary. Instead of connecting straight to the database, the function uses a connection pool to manage connections. This allows multiple concurrent clients to perform operations on the database without having to wait for other clients to finish their transactions. In this

case, when multiple files are being scanned at the same time and the results are being saved to the database, the connection pool ensures that a database connection is always available.
- `CreateTable`: Uses the connection pool to create a new table with the specified name if it does not exist. The table schema includes columns for ID, ObjectKey, BucketName, Result, Antivirus, Timestamp, and VirusType. The SQL query used to execute this operation is available in the appendix at listing 4.
- `Insert`: Uses the connection pool to insert a record into the specified table with values for ObjectKey, BucketName, Result, Antivirus, Timestamp, and VirusType. The SQL query used to insert is available in the appendix at listing 5.

The PostgreSQL instance is provided with a context to close the connection to the database when the application is shutting down.

Aegis' Internal Workflow

s

Metrics and Collectors

The internal metrics collection comprises two primary components: the metric manager and various metric collectors specific to each package. The metric manager is responsible for managing interactions with Prometheus, which includes executing the `Start` and `Stop` methods for handling the starting and graceful shutdown of Prometheus respectively.

Each package contains a metric collector in a file named `metrics.go`. These collectors define the available metrics that can be collected and exported by the respective packages. The `promauto` library facilitates the creation of a global registry when the metric manager is initialized. This registry is accessed by all metric collectors to record the metrics they collect and is also utilized by the Prometheus exporter for publishing these metrics.

Object

The object package presents the `Object` struct as the internal representation of an object within the object store. It includes all methods and attributes related to an object, such as the object key and bucket name. Operations involving an object are performed within the object instance itself.

Since the object represents a concrete entity, there is no need for an interface when using it. This design choice allows the object to have attributes that can be accessed directly, without the need for getter functions.

The `CreateObject` function enables the creation of a new `Object` instance, given a specified object key and bucket name.

The `SetCachePath` method defines the cache path for an object by concatenating the cache path, bucket name, and object key, separated by slashes. This method is called when an update to the cache path is needed.

The `SaveByteStreamToFile` method stores an object's byte stream in a file. First, it checks if the path attribute is empty since, by default, no path is provided. It returns an error if this is the case, as other types of scanners may not always require this information to perform a scan. Next, it ensures that the file's parent directory exists, creating it if necessary. Lastly, the method writes the byte stream to the file using Go's built-in IO writer.

The RemoveFileFromCache method is responsible for deleting an object file from the cache. It tries to remove the file specified by the object's path attribute. If the removal is unsuccessful, it logs an error message and returns the error.

Events Manager

The events package includes the event manager, which is responsible for reading messages from the event queue and forwarding scan requests to the scanner. The Kafka interface provides methods for reading messages from the Kafka queue and closing the connection. The EventsManager struct consists of four fields: a logger, a kafka instance for interacting with Kafka, a scanChan channel to forward scan requests and an eventsCollector for gathering metrics.

The CreateEventsManager function creates a new EventsManager instance, accepting the necessary arguments. The Start method of the EventsManager takes in a context and then enters a loop that uses a switch statement to first check if the context has been canceled. If it has, it closes the scanChan channel, closes the Kafka connection and returns. Otherwise, it invokes the ReadMessage method of the kafka instance to read a message from the Kafka queue. Upon confirming that there is no error and the message is not nil, it increments the eventsCollector counter and creates a new object.Object instance with the received bucket name and object key. It then forwards the object to the scanChan channel for scanning.

Since the event manager runs within a goroutine, if an error occurs, it sends the error to the provided errChan channel.

Object Store

In the object store package, several structs and interfaces are defined to handle object storage operations. The Minio interface contains the abstract object store operations, such as; get, put and remove objects, as well as get and put object tags. These are also reflected by the ObjectStoreCollector in the form of metric counters that track the number of each operation performed.

Once the object store is created by the CreateObjectStore function, it can be used by the rest of the application to perform object storage operations. In addition to the standard object storage operations, two more operations are added to the object store: MoveObject and AddObjectTagging. These both combine multiple standard operations into one as follows:

- MoveObject: Retrieves an object from the source bucket, puts it into the destination bucket, and removes it from the source bucket.
- AddObjectTagging: Retrieves the object tags from the source bucket, adds the new tags to the existing ones, and puts the combined object tags onto the object.

Object Scanner

The scanner package provides the functionality for multiple workflows when it comes to scanning an object. In this instance, an object scanners downloads from the object store, performs a scan with its antivirus engines, and then passes the result to a cleaner which will execute the cleanup policy. Having the ability to use multiple types of scanners allows for flexibility in the system as in the future, the workflow for scanning an object might change. For example, if one of the antivirus engines could require the hash of the file. In this case, another scanner called HashScanner could be created to handle this alternate workflow. For this project, the ObjectScanner will be the only type of scanner implemented.

The `CreateObjectScanner` function creates a new `ObjectScanner` instance with the following arguments:

- `logger`: A logger instance for logging messages.
- `objectStore`: An object store instance for downloading objects.
- `antiviruses`: An array of antivirus instances for scanning objects.
- `cleaner`: A cleaner instance for cleaning up objects.
- `auditLogger`: An audit logger instance for logging scan results.
- `scanCollector`: A scan collector instance for collecting metrics.
- Various configuration values, such as, `removeAfterScan`, `datetimeFormat`, and `cachePath`.

All instances passed to the `CreateObjectScanner` function are interfaced to both allow for future mocking and to allow for other abstract implementations of the interfaces.

The `ScanObject` method handles the workflow for downloading and scanning an object. It takes in an `object.Object` instance, which it fetches from the object store, and an `errChan` for returning errors. It then sets the object cache path by calling `SetCachePath` on the object. With this set, the scanner can perform a `GetObject` on the object store to retrieve the byte stream of the object and call `SaveByteStreamToCache` with the byte stream to save it to the cache.

The scanner can now perform a scan on the object by calling `Scan`, with the cache location, on every antivirus engine. If any of the antivirus engines detect the object as infected, then using the assume the worst mentality, the file is deemed infected. The object is then passed to the cleaner to execute the cleanup policy. During this execution various metrics are being collected by the `scanCollector` about the scan, such as, the number of clean or infected files, total files scanned and total errors encountered. In addition audit logs are also generated by the `auditLogger` for each scan by each antivirus, recording the object key, bucket name, antivirus name, scan result, timestamp and if infected, the virus type.

After performing the scan and dealing with the results, if the `removeAfterScan` flag is set to true, the object is removed from the cache after scanning.

The `ObjectScanner` is will be run within a goroutine, if an error is encountered during the scan, it will be sent to the provided `errChan`.

Cleanup Policies

As mentioned in the previous section, the `ObjectScanner` passes the object to the cleaner to execute the cleanup policy. The cleaner package defines how to react given a clean or infected result from the antivirus engines. Multiple policies are available in the `config.env` file to give the user flexibility in how they want to deal with infected objects. These policies include:

- `Tag`: Adds a tag to the object in the object store based on the scan result.
- `Remove`: Removes the object from the object store if it is deemed infected
- `Quarantine`: Moves the object to a quarantine bucket if it is deemed infected.

In `CreateCleaner` function, a new `Cleaner` instance is created with a logger, object store, metrics and audit loggers, and various configuration parameters such as the cleanup policy and quarantine bucket.

The `Cleanup` method is called by the `ObjectScanner` where is passes the object after it has been scanned. This method uses a switch statement, shown in figure **??**, to determine which policy to implement and executes the appropriate cleanup method. If no policy is specified, the

switch statement has a default cause of logging that it will do nothing, in the case that the user only wants the audit logs. However with a given cleanup policy, the corresponding cleanup method is called. Each of these use the object store and object store to perform the cleanup.

```
switch c.cleanupPolicy {
case "tag":
  err = c.TagInfected(object, result, scanTime)
case "remove":
  err = c.RemoveInfected(object, result, scanTime)
case "quarantine":
  err = c.QuarantineInfected(object, result, scanTime)
default:
  c.logger.Warnln("No cleanup policy found")
}
```

Figure 7: Cleanup policy switch statement

Dispatcher

The `dispatcher` package is responsible for managing the scanning of objects using multiple scanners concurrently. It defines a `Scanner` interface with a single method, `ScanObject`, that will be implemented by one of the available scanners. The `Dispatcher` struct contains three fields: a `logger` for logging messages, a `scanChan` channel for receiving object scan requests, with a `scanners` slice to hold the available scanners.

The `CreateDispatcher` initialises a new `Dispatcher` instance with the required fields. The `Start` method enters into a loop where it ranges over the `scanChan` channel. If the channel is empty, the loop will block until a new object is sent through the channel. If the channel has an object, the dispatcher will spawn a new goroutine to handle the scan. However, if the channel is closed, the loop will continue to process all objects in the channel and then exit (Go by Example, 2022b). This is because when channels are closed, no more values can be sent to them, but the values that have already been sent can still be received (Go by Example, 2022a). This is shown in the provided dispatcher loop code in figure 8.

```
func (d *Dispatcher) Start(errChan chan error, done chan struct{}) {
        var wg sync.WaitGroup

        for request := range d.scanChan {
                for _, scanner := range d.scanners {
                        wg.Add(1)
                        go func(req *object.Object, sc Scanner) {
                                defer wg.Done()
                                sc.ScanObject(req, errChan)
                        }(request, scanner)
                }
        }
        wg.Wait()
        done <- struct{}{} //Send empty done message
}
```

Figure 8: Dispatcher loop

When the program receives a termination signal, there should be no loss of information about incoming scan requests. This is a security risk as it could lead to objects not being scanned. To prevent this, the dispatcher uses a `sync.WaitGroup` to wait for all goroutines to finish before exiting the program. This is done by calling `wg.Add(1)` before starting a new anonymous goroutine to increment an active goroutine counter, and using `defer wg.Done()` when the goroutine has finished to decrement the counter. The `wg.Wait()` call will block until the counter is zero, meaning all goroutines will have finishes processing (Go by Example, 2022c). This ensures that all scans that are currently being processed since `scanChan` was close will be completed before the program exits.

Main

The main package orchestrates the top-level workflow that Aegis executes throughout its operation. The entry point of the program is the `main` function, which performs one task - calling the `run` function and exiting the program based on its return value. This design choice enhances extensibility and testability since alternate workflows can be implemented while maintaining a single entry point. Furthermore, the `run` function can be tested without running the entire program (Ryer, 2020). The `run` function serves as a abstraction from `main`, as it encompasses Aegis' main workflow.

The `run` function is divided into three distinct sections: initialization and configuration, the main loop, and cleanup. The initialization and configuration section is responsible for initializing all the components Aegis requires and configuring them with the values provided by the configuration package.

- The configuration is loaded from `config.env`, and the logger is created. An initial context is also created and passed to everything that requires it, apart from the event system.
- The metric manager and various metric collectors are created, with the metric manager taking in the Prometheus exporter.
- The audit logger is implemented by the PostgreSQL database.
- The object store is implemented by the Minio client.
- The event system is implemented by the Kafka consumer with the `scanChan` passed in as well.
- The scanning workflow is created. This includes creating the antivirus engines, in this case ClamAV, creating the cleaner with the configured policy and passing both of them to the scanners, namely the object scanner. The scanners are then passed to the dispatcher alongside the `scanChan`.

The main loop is the continuous execution of the goroutines that handle Aegis' asynchronous operations. These are the event manager, dispatcher, and metric manager. The goroutines are started using the `go` keyword, which spawns a new goroutine to execute the functions in a separate thread.

An additional context is created and parsed into the `Start` method of the `eventManager`. Multiple channels are then created to handle errors, shutdown command, and the shutdown complete with `errChan`, `shutdownChan` and `done` respectively. The `Start` methods of the `eventManager`, `dispatcher` and `metricManager` are called to begin the main workflow of the program.

An error channel (`errChan`) is created to handle errors generated by the event manager and metric manager goroutines.

Finally, the shutdown section ensures a smooth termination when the program receives an interrupt signal or encounters errors from the goroutines. The shutdown sequence is vital as it allows Aegis to maintain the progress of processed objects when receiving messages, enabling it to resume scanning objects from where it left off upon restart. The shutdown sequence unfolds as follows:

When an interrupt signal or an error from any goroutine is received, Aegis starts its graceful shutdown sequence. A select statement is used to wait for a message from either the `errChan` or `shutdownChan` channels. If any of these channels receive a message, a message or error is logged, and the shutdown sequence begins by canceling the context passed to the event manager. This action halts the event manager from consuming messages from the Kafka consumer and subsequently closes the `scanChan` channel. As a result, incoming notifications remain in the Kafka queue and can be consumed by Aegis upon restart leading to no scans lost.

The code then waits for the `done` channel to send a message, signaling that the goroutines have completed processing the remaining objects in the `scanChan` channel. Once this process is finished, the program stops the Prometheus metric exporter and exits with a status code of 0, indicating a successful operation.

Testing

Mocking

Unit Tests

Kubernetes Deployment

Docker

Aegis Containerisation

Kubernetes

Helm

K3d

6. Results and Evaluation

7. Product Issues / Future work

8. Conclusions

9. Reflection on Learning

10. Appendix

Listing 1: Example Kafka Notification


```
{
  "EventName": "s3:ObjectCreated:Put",
  "Key": "test-bucket/gantt.png",
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "minio:s3",
      "awsRegion": "",
      "eventTime": "2023-03-24T12:23:27.844Z",
      "eventName": "s3:ObjectCreated:Put",
      "userIdentity": {
        "principalId": "minioadmin"
      },
      "requestParameters": {
        "principalId": "minioadmin",
        "region": "",
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "content-length": "0",
        "x-amz-id-2": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
        "x-amz-request-id": "174F5A6C715ECB50",
        "x-minio-deployment-id": "439d9d33-3cca-42ae-b778-d703cbf9bbf7",
        "x-minio-origin-endpoint": "http://192.168.1.138:9000"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "Config",
        "bucket": {
          "name": "test-bucket",
          "ownerIdentity": {
            "principalId": "minioadmin"
          },
          "arn": "arn:aws:s3:::test-bucket"
```

```
        },
        "object": {
          "key": "gantt.png",
          "size": 92704,
          "eTag": "22c4ae87e652bce0865536964cb8bb8d",
          "contentType": "image/png",
          "userMetadata": {
            "content-type": "image/png"
          },
          "sequencer": "174F5A6C71C977E8"
        }
      },
      "source": {
        "host": "127.0.0.1",
        "port": "",
        "userAgent": "MinIO (darwin; arm64) minio-go/v7.0.49 MinIO Console/(dev)"
      }
    }
  ]
}
```


Listing 2: Example Kafka Notification

```
/Users/username/test.jpg: OK

----------- SCAN SUMMARY -----------
Infected files: 0
Time: 0.332 sec (0 m 0 s)
Start Date: 2023:03:27 11:53:15
End Date:   2023:03:27 11:53:16
```

Listing 3: Example Exposed Metrics

```
# HELP aegis_kafka_total_messages Kafka total messages received
# TYPE aegis_kafka_total_messages counter
aegis_kafka_total_messages 0
# HELP aegis_objectstore_get_objects Object Store total get objects
# TYPE aegis_objectstore_get_objects counter
aegis_objectstore_get_objects 0
# HELP aegis_objectstore_get_objects_tagging Object Store total get objects tagging
# TYPE aegis_objectstore_get_objects_tagging counter
aegis_objectstore_get_objects_tagging 0
# HELP aegis_objectstore_put_objects_tagging Object Store total put objects tagging
# TYPE aegis_objectstore_put_objects_tagging counter
aegis_objectstore_put_objects_tagging 0
# HELP aegis_scanner_clean_files Total of clean files scanned by Aegis
# TYPE aegis_scanner_clean_files counter
aegis_scanner_clean_files 0
# HELP aegis_scanner_errors Total number of errors encountered during scans by Aegis
# TYPE aegis_scanner_errors counter
aegis_scanner_errors 0
# HELP aegis_scanner_infected_files Total of infected files scanned by Aegis
```

```
# TYPE aegis_scanner_infected_files counter
aegis_scanner_infected_files 0
# HELP aegis_scanner_time Time taken to perform a scan
# TYPE aegis_scanner_time histogram
aegis_scanner_time_bucket{le="0"} 0
aegis_scanner_time_bucket{le="125"} 0
aegis_scanner_time_bucket{le="250"} 0
aegis_scanner_time_bucket{le="500"} 0
aegis_scanner_time_bucket{le="1000"} 0
aegis_scanner_time_bucket{le="2000"} 0
aegis_scanner_time_bucket{le="4000"} 0
aegis_scanner_time_bucket{le="8000"} 0
aegis_scanner_time_bucket{le="16000"} 0
aegis_scanner_time_bucket{le="+Inf"} 0
aegis_scanner_time_sum 0
aegis_scanner_time_count 0
# HELP aegis_scanner_total_scans Total number of scans performed by Aegis
# TYPE aegis_scanner_total_scans counter
aegis_scanner_total_scans 0
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycle
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 17
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.19.4"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 1.363496e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 1.363496e+06
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 4941
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 2757
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metada
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 3.745832e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 1.363496e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
```

```
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 3.678208e+06
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 3.8912e+06
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 16646
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 3.678208e+06
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 7.569408e+06
# HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage coll
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 0
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 19403
# HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.
# TYPE go_memstats_mcache_inuse_bytes gauge
go_memstats_mcache_inuse_bytes 12000
# HELP go_memstats_mcache_sys_bytes Number of bytes used for mcache structures obtained f
# TYPE go_memstats_mcache_sys_bytes gauge
go_memstats_mcache_sys_bytes 15600
# HELP go_memstats_mspan_inuse_bytes Number of bytes in use by mspan structures.
# TYPE go_memstats_mspan_inuse_bytes gauge
go_memstats_mspan_inuse_bytes 95184
# HELP go_memstats_mspan_sys_bytes Number of bytes used for mspan structures obtained fro
# TYPE go_memstats_mspan_sys_bytes gauge
go_memstats_mspan_sys_bytes 97632
# HELP go_memstats_next_gc_bytes Number of heap bytes when next garbage collection will t
# TYPE go_memstats_next_gc_bytes gauge
go_memstats_next_gc_bytes 4.194304e+06
# HELP go_memstats_other_sys_bytes Number of bytes used for other system allocations.
# TYPE go_memstats_other_sys_bytes gauge
go_memstats_other_sys_bytes 1.300043e+06
# HELP go_memstats_stack_inuse_bytes Number of bytes in use by the stack allocator.
# TYPE go_memstats_stack_inuse_bytes gauge
go_memstats_stack_inuse_bytes 819200
# HELP go_memstats_stack_sys_bytes Number of bytes obtained from system for stack allocat
# TYPE go_memstats_stack_sys_bytes gauge
go_memstats_stack_sys_bytes 819200
# HELP go_memstats_sys_bytes Number of bytes obtained from system.
# TYPE go_memstats_sys_bytes gauge
go_memstats_sys_bytes 1.3552656e+07
# HELP go_threads Number of OS threads created.
# TYPE go_threads gauge
go_threads 12
```

```
# HELP promhttp_metric_handler_requests_in_flight Current number of scrapes being served.
# TYPE promhttp_metric_handler_requests_in_flight gauge
promhttp_metric_handler_requests_in_flight 1
# HELP promhttp_metric_handler_requests_total Total number of scrapes by HTTP status code
# TYPE promhttp_metric_handler_requests_total counter
promhttp_metric_handler_requests_total{code="200"} 0
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
```

Listing 4: Create Table SQL Query

```
CREATE TABLE IF NOT EXISTS %s (
        ID SERIAL PRIMARY KEY,
        ObjectKey TEXT NOT NULL,
        BucketName TEXT NOT NULL,
        Result TEXT NOT NULL,
        Antivirus TEXT NOT NULL,
        Timestamp TIMESTAMP NOT NULL,
        VirusType TEXT
);
```

Listing 5: Insert into Table SQL Query

```
INSERT INTO %s (ObjectKey,
                BucketName,
                Result,
                Antivirus,
                Timestamp,
                VirusType) VALUES ($1,
                                   $2,
                                   $3,
                                   $4,
                                   $5,
                                   $6
)
// Where %s = tableName
```

References

2022. [Accessed: 24 Mar 2023].

ClamAV. 2023. Clamav, Available at: https://www.clamav.net/. [Accessed: 24 Mar 2023].

Go by Example. 2022a. Closing channels, Available at: https://gobyexample.com/closing-channels. [Accessed: 30 Mar 2023].

Go by Example. 2022b. Ranging over channels, Available at: https://gobyexample.com/range-over-channels. [Accessed: 30 Mar 2023].

Go by Example. 2022c. Waitgroups, Available at: https://gobyexample.com/waitgroups. [Accessed: 30 Mar 2023].

Levy, E. 2022. Kafka vs rabbitmq: Architecture, performance, and use cases, Available at: https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case. [Accessed: 24 Mar 2023].

M, I. 2022. Installing apache kafka without zookeeper: Easy steps 101, Available at: https://hevodata.com/learn/kafka-without-zookeeper/. [Accessed: 24 Mar 2023].

McKusick, M. K., Neville-Neil, G. V. and Watson, R. N. M. 2014. *Process Management in the FreeBSD Operating System*, Addison-Wesley Professional, p. 928. 2nd ed., Available at: https://www.informit.com/articles/article.aspx?p=2249436&seqNum=8.

MinIO. 2023a. Minio go sdk repository, Available at: https://github.com/minio/minio-go. [Accessed: 15 Mar 2023].

MinIO. 2023b. Minio object storage, Available at: https://min.io/. [Accessed: 15 Mar 2023].

MinIO Erasure Coding. 2023. Minio erasure coding, Available at: https://min.io/docs/minio/linux/operations/concepts/erasure-coding.html. [Accessed: 25 Apr 2023].

Ryer, M. 2020. Why you shouldn't use func main in go, Available at: https://pace.dev/blog/2020/02/12/why-you-shouldnt-use-func-main-in-golang-by-mat-ryer.html. [Accessed: 30 Mar 2023].

Srinivasan, G., Eidelman, A. and Casmer, E. 2022. Integrating amazon s3 malware scanning into your application workflow with cloud storage security, Available at: https://aws.amazon.com/blogs/apn/integrating-amazon-s3-malware-scanning-into-your-application-workflow-with-cloud-storage-security/. [Accessed: 21 Mar 2023].

uber go. 2023. Zap, Available at: https://github.com/uber-go/zap. [Accessed: 25 Mar 2023].