Google Summer of Code

PROPOSAL

KubeArmor Observability and Policy Discovery Helper Tool

By

Nathaniel Jason

CLOUD NATIVE COMPUTING FOUNDATION

kubearmor

Table Of Contents

1. Abstract

Kubernetes is an open-source system that is used for automating deployment, scaling, and management of containerized applications. One of the hardest challenges of running Kubernetes is to provide security. KubeArmor is an open-source tool that provides policy based systems to restrict unwanted and malicious behavior of cloud-native workloads at runtime by utilizing eBPF and Linux Security Modules. There would still be a challenge especially for new users to create the most optimal security policies for their own use cases to utilize KubeArmor effectively.

2. Background

2.1. Problem

Running Kubernetes should be done with the utmost security in mind. KubeArmor provides a policy based system to increase the quality of security in a Kubernetes cluster. Users need to define their own policy when implementing KubeArmor. Most users might find it challenging to define the most appropriate and effective security policy based on their use case. Currently there are limited ways for users to actually get observability data that can be used to assist them and provide better reasoning when creating an optimal security policy.

Related issues: https://github.com/kubearmor/KubeArmor/issues/613

2.2. Solution

The solution to this problem is by providing visibility telemetry events to show pod or container observability data. Observability data can consist of processes executions, file system accesses, and network accesses. This information is expected to assist users to create an optimal security policy. This solution is implemented by developing and deploying a Kubernetes service that will connect to the KubeArmor DaemonSet and get the events from the daemonsets. These events will then be aggregated and stored in the database for future use. Users can get the data by using the KubeArmor CLI program which will be extended to accommodate this solution.

2.3. Benefit

2.3.1. KubeArmor Users

KubeArmor users will have the ability to get observability data which they can utilize to create an optimal policy for their use cases.

2.3.2. KubeArmor

This project will indirectly increase KubeArmor user count by making it more beginner friendly. This can be achieved because users that don’t know how to make an optimal security policies for their pods now have an assist in the form of aggregated and summarized data that is presented in the most user-friendly way. KubeArmor will get an additional feature that allows KubeArmor to get observability data and store it in a centralized place. This feature has a high possibility to be a foundation to future features for example a recommendation engine for security policy.

2.3.3. Google

This project will indirectly extend the security quality of Kubernetes, one of the biggest and most used open-source projects which originate from Google.

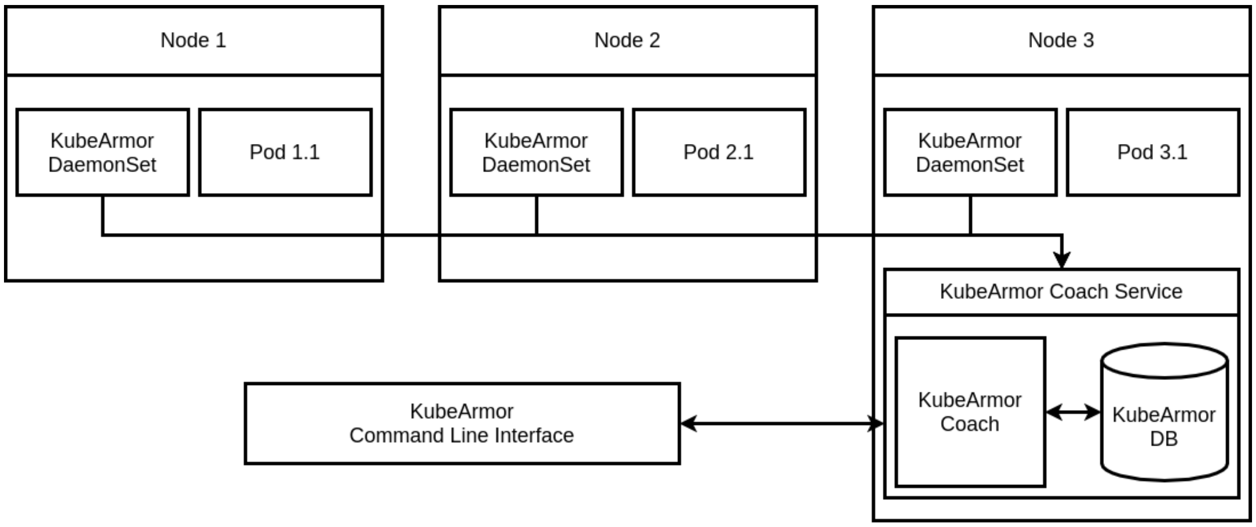
3. Project Details

3.1. User Story and Acceptance Criteria

These are the user story and acceptance criteria for this project.

- As a user, I am able to get observability data.
    - Observability data can consist of processes active in the memory, file system paths that are accessed, network calls used, and system call maps.
    - Observability data that is retrieved can be on namespace level, pod level or at container level, across multi-node deployments.
    - Observability data that is shown can be in complete form or simple form.
    - Observability data will be stored in a database.
    - Observability data can be viewed on text-based user interface by using the KubeArmor CLI.

3.2. System Design

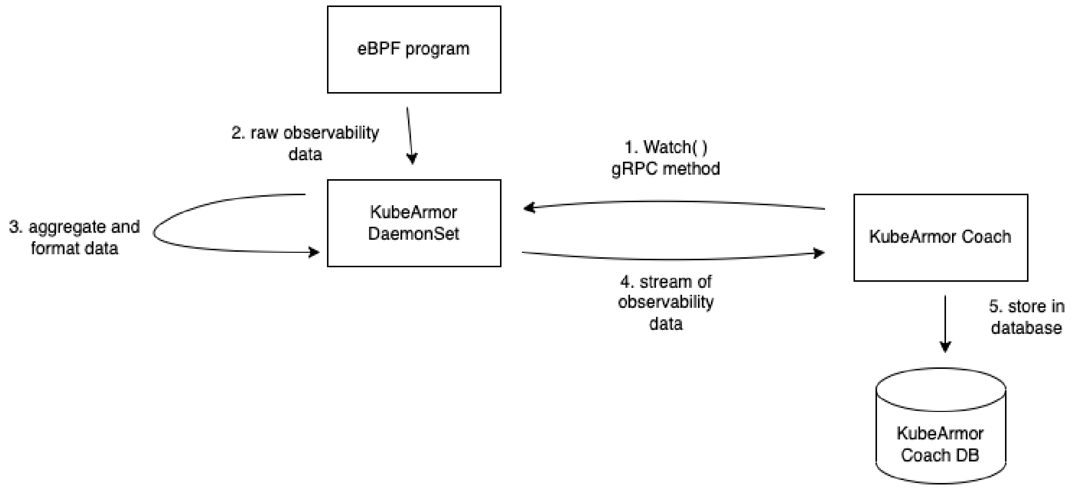


- KubeArmor DaemonSet will communicate with the KubeArmor Coach through GRPC APIs.
- KubeArmor Coach will request each KubeArmor DaemonSet to stream the observability data as response.

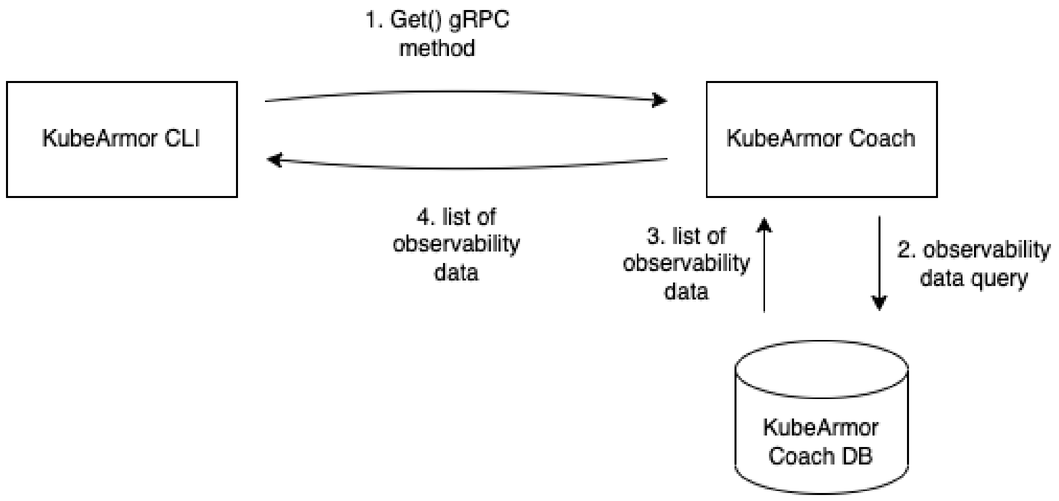
- KubeArmor CLI will communicate with KubeArmor Coach using GRPC APIs, the information will then be displayed through text-based user interface.
- All of the complex logic and algorithm will be done on the KubeArmor Coach service, other components such as KubeArmor DaemonSet or KubeArmor CLI will only send or receive data.

3.3. Data Flow

3.3.1. Watch and Store Observability Data



3.3.2. Get Observability Data


1. Get() gRPC method
KubeArmor CLI
KubeArmor Coach
4. list of observability data
3. list of observability data
2. observability data query
KubeArmor Coach DB

3.4. Implementation

3.4.1. KubeArmor DaemonSet

3.4.1.1. Protobuf Definition

We will be extending the current KubeArmor DaemonSet gRPC server to also include a method that we will need to use to send data to KubeArmor Coach. This protobuf definition includes service and message definition.

Service Definition

```
service ObservabilityService {
    rpc Watch(SaveRequest) returns (stream SaveResponse);
}
```

We need to extend the current KubeArmor DaemonSet gRPC server. There is one method that we need to add, the *Watch* method. The client of this method would be the KubeArmor Coach service. The KubeArmor DaemonSet then would return the response in the form of a stream of data. Technically, this stream or connection for this response would not be closed unless the KubeArmor DaemonSet pod is terminated.

Message Definition

```
message FileAccessData {
    string path = 1;
}

message ProcessSpawnData {
    string user = 1;
    string command = 2;
}

message NetworkCallData {
    string protocol = 1;
    string address = 2;
}

message ObservabilityData {
    string namespace_id
    string deployment_id
    string node_id
    string pod_id
    string container_id
    string operation_type
    Timestamp created_at
    oneof operation_data {
        FileAccessData file
        ProcessSpawnData process
        NetworkCallData network
    }
}
```

This is the data structure for general observability data and the operation-specific data. This will be explained more in the 3.4.2.1 section where we are talking about KubeArmor Coach.

```
message SaveRequest {
}

message SaveResponse {
    ObservabilityData data = 1;
}
```

3.4.2. KubeArmor Coach

3.4.2.1. Protobuf Definition

Since we will be implementing the KubeArmor Coach service as a gRPC server, we will need to define the proto file. Proto file usually consist of two parts, the message definition and service definition.

Service Definition

```
service ObservabilityService {
    rpc Get(GetRequest) returns (GetResponse);
}
```

There will be one service, *ObservabilityService*, which will be used by KubeArmor CLI and KubeArmor DaemonSet as the client, and will be implemented in KubeArmor Coach service as the server. *ObservabilityService* will have one method, *Get*. The *Get* method will be used by KubeArmor CLI to get observability data that has been stored.

Message Definition

```
message FileAccessData {
    string path = 1;
}

message ProcessSpawnData {
    string user = 1;
    string command = 2;
}

message NetworkCallData {
    string protocol = 1;
    string address = 2;
}
```

There are three types of operation that will be stored in the KubeArmor Coach database, file accesses, processes spawn, and network call. We will define the data structure for each of this operation. We need to make separate different data structures so that each operation has a more flexible data definition. If we need to add another data to one of the operations, we can just add another field to that operation data structure. If we were to make only one data structure that can be used by all operations, it would be harder and there would be many unused fields depending on the operation.

```
message ObservabilityData {
    string namespace_id
    string deployment_id
    string node_id
    string pod_id
    string container_id
    string operation_type
    Timestamp created_at
    oneof operation_data {
        FileAccessData file
        ProcessSpawnData process
        NetworkCallData network
    }
}
```

This is the data structure that we will receive from KubeArmor DaemonSet and store them in the database. As we can see, this data structure consists of metadata and specific operation data. This type of data structure can be used because the metadata for all the operations is the same, such as namespace ID, deployment ID, node ID and etc. By using this data structure, we can have general and specific data at the same time. The specific data type can be achieved by utilizing the *oneof* syntax in protobuf. This syntax lets us have a variable associated with 3 possible data types. If we were to add another type of operation in the future, we can just add another data type in the *oneof* clause, no need much modification for extending the feature. This will ensure scalability.

```
message GetRequest {
    string namespace_id = 1;
    string deployment_id = 2;
    string node_id = 3;
    string pod_id = 4;
    string container_id = 5;
    string operation_type = 6;
    Duration time = 10;
    string operation_data_filter = 11;
}

message GetResponse {
    repeated ObservabilityData data = 1;
}
```

This is the data structure for request and response in the RPC method *Get*, which will be used by the KubeArmor CLI. The response for this method will be an array of *ObservabilityData*. Array implementation in protobuf can be achieved by utilizing the syntax *repeated*. The request data structure for the *Get* method is filled by the filters or queries that we want to have in our command. There is one filter that is a bit different than the others, the *operation_data_filter*. This field is used as the query string for specific operation data. For example, if we want to search for a specific file path in the file access operation, *operation_data_filter* for example will have a value

```
path=/dev/myfile.txt
```

This string can consist of multiple filters by separating it with a semicolon, for example,

```
user=root;command=/bin/some-command
```

In the future, it is really possible if we want to add a logic parser for this string, we can use the operation ‘and’ and ‘or’, for example,

```
user=root||(user=some-user&&command=/bin/some-command)
```

3.4.2.2. Database Design

This proposal will provide 3 approaches for the database design. We will examine each design, analyze the advantages and disadvantages of each approach, and in the end choose one of the approaches.

Approach 1: Single Table For All Operation

```sql
CREATE TABLE Observabilities (
    ID VARCHAR(255),
    NamespaceID VARCHAR(255),
    DeploymentID VARCHAR(255),
    NodeID VARCHAR(255),
    PodID VARCHAR(255),
    ContainerID VARCHAR(255),
    OperationType VARCHAR(255),
    Details TEXT,
    CreatedAt TIMESTAMP,

    PRIMARY KEY (ID)
);
```

In the first approach, we will only need one table to store the data. This table consists of the observability metadata and the specific operation data. The specific operation data is stored in the *Details* field. The operation-specific data is stored as stringified JSON data. This will ensure scalability in the operation-specific data structure. If we want to add another field in the operation-specific data structure. We don't need to add another column or even a table. We can also extend to store another operation type without modifying the table.

Using this data type to store operation-specific data also supports our basic *operation_data_filter*. We can just check the *Details* string consists of the *operation_data_filter* string separated by the delimiter. Here is the code snippet for Go programming language with the help of Gorm ORM library.

```
// ...
delimiter := “;”
tempDetailsFilter := string.split(req.OperationDataFilter, delimitter)

detailsFilter := []string{}
for _, filter := range tempDetailsFilter {
    filterDelimitter := “=”
    filterSplit := string.split(filter, filterDelimitter)

    // trying to replicate on how it is stored in JSON
    finalFilter := fmt.Sprintf(`”%s”:”%s”`, filterSplit[0], filterSplit[1])
    // for example, filter with value “path=some-path” will produce
    // finalFilter with value of `”path”:”some-path”`

    detailsFilter = append(detailsFilter, finalFilter)
}

// initialize GORM query object
query := db.Tables(“observabilities”)

for _, filter := detailsFilter {
    // construct string with sql syntax that will check for substring
    sqlFilterString = "%" + filter + "%"

    query = query.Where("details LIKE %", sqlFilterString)
}

// ...
```

As we can see, it is really possible to implement the first approach in the KubeArmor Coach service in Go programming language. If we want to extend to allow logical syntax such as “and” and “or”, it will still be really possible, it’s just that we need an advanced parser algorithm implemented in KubeArmor Coach service.

Advantage	Disadvantage
1. In most cases, the fastest compared to the other approaches because we only query from one table. 2. High scalability in operation-specific data and operation type, we can extend them without much modification on the database. 3. Only need to maintain one table, if we want to add other metadata would be easy.	1. Might be hard if we want to extend our operation-specific data filter to implement logical syntax such as “and” and “or”.

Approach 2: Single Table For Metadata, Multiple Table For Operation

```
CREATE TABLE Observabilities (
    ID VARCHAR(255),
    NamespaceID VARCHAR(255),
    DeploymentID VARCHAR(255),
    NodeID VARCHAR(255),
    PodID VARCHAR(255),
    ContainerID VARCHAR(255),
    OperationType VARCHAR(255),
    DetailsID VARCHAR(255),
    CreatedAt TIMESTAMP,

    PRIMARY KEY (ID)
    -- need to have foreign key that references to few possible tables
);

CREATE TABLE Files (
    ID VARCHAR(255),
    Path VARCHAR(255)

    PRIMARY KEY (ID)
)

CREATE TABLE Processes (
    ID VARCHAR(255),
    User VARCHAR(255),
    Command VARCHAR(255),

    PRIMARY KEY (ID)
)

CREATE TABLE Networks (
    ID VARCHAR(255),
    Protocol VARCHAR(255),
    Address VARCHAR(255),
    State VARCHAR(255),

    PRIMARY KEY (ID)
)
```

In the second approach, we will mainly use two types of table, one table to store the metadata of operation and the other type of table to store operation-specific data. Since we have 3 types of operations, we will create one table for each operation.

This approach will be needing the same amount of effort to implement an operation-specific data filter because we need to implement some kind of algorithm to parse the filter string and translate it to Gorm or even raw SQL query.

Most likely we will not choose this approach since this approach is not applicable in SQL. On *Observabilities* table, there should be a foreign key that refers to either one of the operation tables. The foreign key is an important constraint that we need to implement to ensure data consistency. This kind of reference is called Polymorphic Associations, where the foreign key column contains an id value that must exist in one of a set of target tables. Currently, SQL does not support this kind of constraint.

This approach also will be the slowest than the other two approaches since we need to join the tables and query them 3 times, one time for each operation.

Advantage	**Disadvantage**
1. Medium scalability, if we want to add another field in the operation metadata would be easy since we only maintain one table.	1. Need to drop off the foreign key constraint since polymorphic association is not supported in SQL. 2. In most cases, it is the slowest

2. More structured database for each operation.	compared to the others because we have to join and query 3 times, one for each operation.

Approach 3: Separate Table For Each Operation

```
CREATE TABLE Files (
    ID VARCHAR(255),
    NamespaceID VARCHAR(255),
    DeploymentID VARCHAR(255),
    NodeID VARCHAR(255),
    PodID VARCHAR(255),
    ContainerID VARCHAR(255),
    OperationType VARCHAR(255),
    CreatedAt TIMESTAMP,

    Path VARCHAR(255)

    PRIMARY KEY (ID)
)

CREATE TABLE Processes (
    ID VARCHAR(255),
    NamespaceID VARCHAR(255),
    DeploymentID VARCHAR(255),
    NodeID VARCHAR(255),
    PodID VARCHAR(255),
    ContainerID VARCHAR(255),
    OperationType VARCHAR(255),
    CreatedAt TIMESTAMP,
    User VARCHAR(255),
    Command VARCHAR(255),

    PRIMARY KEY (ID)
)

CREATE TABLE Networks (
    ID VARCHAR(255),
    NamespaceID VARCHAR(255),
    DeploymentID VARCHAR(255),
    NodeID VARCHAR(255),
    PodID VARCHAR(255),
    ContainerID VARCHAR(255),
    OperationType VARCHAR(255),
    CreatedAt TIMESTAMP,

    Protocol VARCHAR(255),
    Address VARCHAR(255),
    State VARCHAR(255),

    PRIMARY KEY (ID)
)
```

In this third approach, we will be storing the metadata of operation data in each operation table. We will maintain a total of 3 tables, one for each operation. This will have a scalability problem if we want to add more fields to the metadata. We need to apply changes to 3 tables.

This approach will be needing the same amount of effort to implement an operation-specific data filter because we need to implement some kind of algorithm to parse the filter string and translate it to Gorm or even raw SQL query.

This approach will also be slower than the first approach since we need to query 3 times, one for each operation. It is still faster than the second approach since we don't need to join the tables.

Advantage	Disadvantage
1. More structured database for each operation.	1. Less scalable, we need to maintain three tables that have a similar structure. A structure modification on the metadata needs to be implemented on three tables. 2. In most cases, it is slower compared to the first approach because we have to query 3 times, one for each operation.

Decision

The approach that will be chosen for database design is **Approach 1**. The first approach is the fastest and has the highest scalability. The disadvantage of the first approach is not really critical since the disadvantage is on the development side, most likely a one-time effect. Whereas on the other approach the disadvantages are actually more critical since it is on the performance side, a continuous effect.

3.4.2.3. Technical Approach

3.4.2.3.1. Limited Database Storage Handling

Currently, we only use one database to store all of the observability data. If we use this in a local or staging environment it would not be a problem since the observability data will not be that much. If we use this in a production environment, there is a high chance that we will deal with a lot of observability data in a short period of time. There would be no space left in the database if we don't provide a mechanism to anticipate this.

Approach 1: SQL Event

In this approach, we will handle this problem by deleting existing records from the database based on the time. For example, we will delete all records that are older than 30 days. We will do the checking and deleting every period of time, for example every day. We can use events in SQL Here is the example implementation of events in MySQL.

```
CREATE EVENT delete_old_records
  ON SCHEDULE
    EVERY 1 DAY
    STARTS (TIMESTAMP(CURRENT_DATE) + INTERVAL 1 DAY + INTERVAL 1 HOUR)
  DO
    -- query to delete old records
```

As we can see, the implementation is pretty easy, we just need to add a script. One of the downsides of this approach is that there is not much modification that can be done. We can only manipulate the data in the database tables. The only threshold or parameters that we can set are time, which most likely is not the best parameters. This is because every setup can have different velocity of logs produced, some setup may only need one week to actually fill the database storage to full, some even need longer. Using parameters with a fixed value is not a really good idea.

Advantage	**Disadvantage**
1. Relatively simple implementation.	1. Limited choice for improvement. Currently we can only use time as a threshold. The threshold that we can define is also fixed or constant. 2. .No much room for extension or modification. We can only manipulate tables in the database.

	3. With the current limitation, it might not be the most optimal solution.

Approach 2: Cronjob

A Cronjob creates jobs on a repeating schedule. We can set a schedule and a set of commands to be executed in cronjob. Kubernetes provides Cronjob resource. Here is the implementation using Kubernetes resource.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "0 0 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: test-image:latest
            imagePullPolicy: IfNotPresent
            command:
            - /some-script-to-delete-old-records
          restartPolicy: OnFailure
```

We can specify the schedule from the *spec.schedule* value. The value “0 0 * * *” value means that the job will run everyday at 12 AM. This approach will have much more flexibility and room for modification and improvement. We can make any scripts here to handle this problem.

For this proposal, we are only going to create a script that deletes old records and then run those scripts using Cronjob. In the future, it does not close the possibility that we can implement a better performance script to handle this problem.

Advantage	Disadvantage
1. Relatively simple implementation. 2. Has a lot of room for improvement and modification to boost performance.	1. Maybe not the best performance to handle this problem using the current script to delete old records.

Approach 3: Smart Tiering

This approach is different from the other 2 approaches before. In this approach we are trying to archive whereas the other 2 approaches try to remove records. Basically, not all data provide the same value to the user. Some may provide more value than the others. Therefore we must agree that not all data is made equal. Keeping data comest at a cost, a storage cost. When our observability data grows at a huge rate we might need to provide more storage to accept and accommodate all those data.

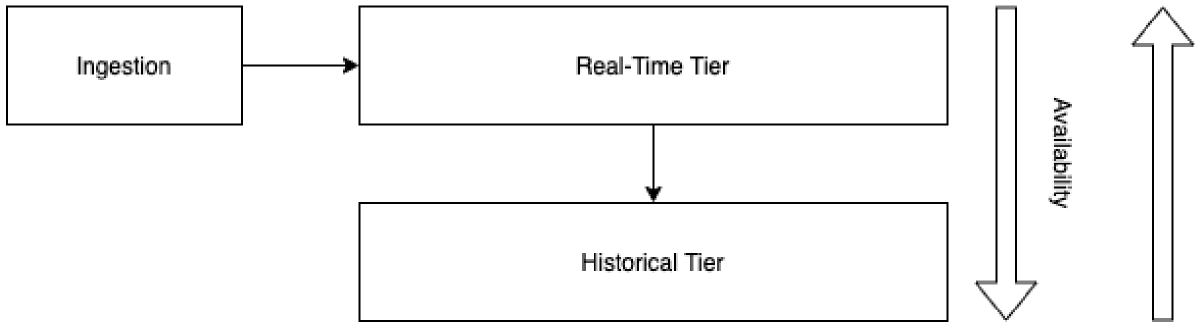
If we look at common operations practices, data tend to get accessed far less when they aged. Our observability data most likely will follow this principle. Old data is far less important than newly saved data. This doesn't mean that aged data is not needed and provides no value, it is just the value is far less. We still keep storing the old data because there is no compromise on data integrity. We can group our data into three categories or tier.

- Critical data which is frequently used
- Active data which is infrequently used
- Historical data which is seldom used

Since not all data provide the same value, therefore not all data should cost the same. We can also group storage into three tiers based on their cost and availability.

- Real-Time Tier for operational data that will be accessed in real time with redundant hot replicas with really high availability and the most expensive cost. Failure in the system won't be a problem when we have hot replicas.
- Smart Tier for infrequent querying, have the same real time performance, backed up with cold replicas but just with a little bit less availability and less expensive than the Real-Time Tier.
- Historical Tier which will be used mostly for compliance and auditing. The example for this storage is AWS S3 or Azure Blob which is only used for archiving. This storage cost far less than the other two tier but it comes with far less availability too.

These are the best practices of using the smart-tiering approach. We will modify this solution to be suitable with the KubeArmor use case. We don't really need to implement the hot replicas on Real-Time Tier since we can tolerate a decrease in availability a little bit for the read operation from the database. We also don't need to implement a Smart Tier; however we can just go directly to the Historical Tier. Here is the final look for our approach.



When our observability data are inserted, it will go directly to the Real-Time Tier. Occasionally from time to time we need to archive data in the Real-Time Tier to be stored in the Historical Tier. This can be achieved by using the Cronjob similar to the second approach. We can create a script that archives data from Real-Time Tier to Historical Tier which will be executed daily or every some period of time.

Advantage	Disadvantage
1. Ensures data integrity and no loss of data. 2. Efficient cost strategy.	1. Might be harder to implement compared to the other two approaches.

Decision

The approach that we choose is **Approach 2: Cronjob** combined with **Approach 3: Smart Tiering.** We will start by the second approach, implementing a cronjob that executes a script to delete old records. After that we can improve the script to do smart tiering.

3.4.2.3.2. Discovering All DaemonSet Pod

Actually the communication pattern between the KubeArmor Coach and KubeArmor DaemonSet is not one to one. The KubeArmor Coach needs to communicate with all the KubeArmor DaemonSet pods. We can achieve this pretty easy using the Kubernetes API. We can just use the "Get Pods" API.

```
GET /api/v1/namespaces/{namespace}/pods
```

We also need to specify the *labelSelector* parameter to target the KubeArmor DaemonSet pods. According to this deployment yaml file in the repo, we can just use this label.

```
kubearmor-app: kubearmor
```

This API will return an array of pod which includes the IP address information that we can use to make the gRPC call.

3.4.3. KubeArmor Command Line Interface

KubeArmor CLI will communicate with KubeArmor Coach service through gRPC call. We will also extend the *observe* command from KubeArmor CLI or *karmor*. This will be the general syntax of the *observe* command.

```
karmor observe <resource_name> <resource_id> [flags]

resource_name:
Value can be namespace, deployment, pod, container. If the resource_name is
plural then we assume it observe all of them. If the resource_name is
singular and if it is followed by the resource_id, we assume that is a
specific observation

flags:
  • -D, --describe : this will show the data similar to the kubectl
    describe format where there would be information about the resource
    first and then the observability data
  • -o, --operation : the operation that we want to observe. The available
    values are file|network|process
  • -t, --time : the time observed from current time. The example value is
    “5d” if we want to display the data for the last five days. This can
    follow the syntax “Y”, “M”, “w”, “d”, “h”, “m”, “s”, representing
    years, months, weeks, days, hours, minutes and seconds.
  • -f, --filter : the filter for operation-specific data, multiple filter
    should be separated with “;” as delimitter
  • -n, --namespace : the namespace that we want to observe on. Set to
    “default” if not specified.
  • -c, --container : the container name
```

These are the few examples of using the *karmor observe* command.

1. Observe all file access on pod level.

```
$ karmor observe pods -o file

NAME        PATH                       TIME
my-pod-1    /bin/dev/some-file.txt     2022-05-20 19:10:05
```

```
my-pod-3   /bin/dev/that-file.txt   2022-05-18 15:01:32
my-pod-2   /bin/dev/this-file.txt   2022-05-17 01:11:01
...
```

2. Observe all processes spawn by *root* user on my-node node.

```
$ karmor observe node my-node -o process -f user=root

NAME       USER   COMMAND        TIME
my-node    root   systemd        2022-05-20 19:10:05
my-node    root   migration/1    2022-05-19 09:52:15
my-node    root   this-command   2022-05-19 12:16:54
...
```

3. Observe all network access using TCP protocol on con-1 container on pod-1 pod and show it on "describe" format.

```
$ karmor observe pod pod-1 -d -o network -f protocol=TCP -c con-1

NAMESPACE: default
NODE:      node-1
POD:       pod-1
CONTAINER: con-1

PROTOCOL   ADDRESS          TIME
tcp        1.2.3.4          2022-05-20 19:10:05
tcp        192.168.100.1    2022-05-19 09:52:15
tcp        216.239.38.120   2022-05-19 12:16:54
...
```

We will later extend the output of KubeArmor CLI to use a test-based user interface. There is a lot of room for extension when transforming to text-based user interface. We can have a more interactive type of output or even display graphs and charts to help users consume the information better.

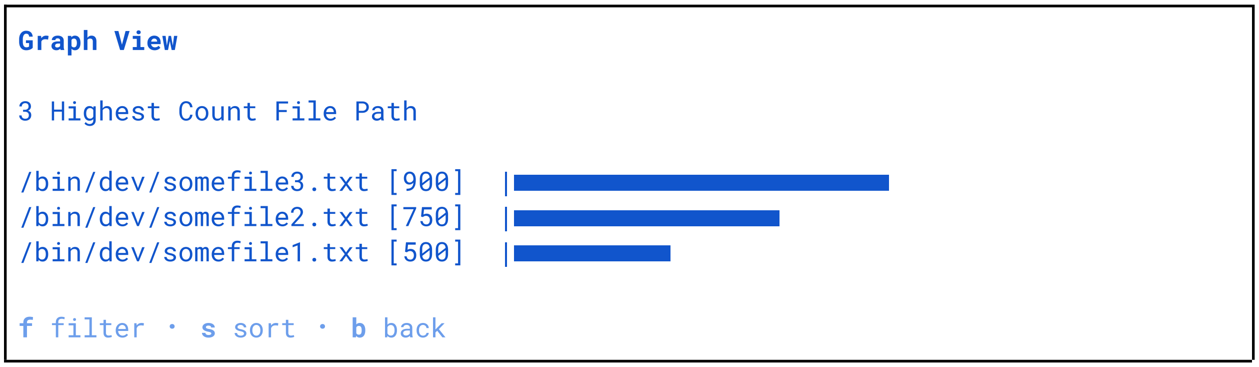
```
$ karmor observe node my-node -o process
```

```
NAME       USER    COMMAND         TIME
my-node    root    systemd         2022-05-20 19:10:05
my-node    root    migration/1     2022-05-19 09:52:15
my-node    root    this-command    2022-05-19 12:16:54
...

f filter · s sort · g graph
```

```
Filter By
[X] NAME
[ ] USER
[ ] COMMAND

b back · s sort · g graph
```



```
Graph View

3 Highest Count File Path

/bin/dev/somefile3.txt [900]  |
/bin/dev/somefile2.txt [750]  |
/bin/dev/somefile1.txt [500]  |

f filter · s sort · b back
```

There is lots of room for extension for our text-based user interface. We can make more interactive program for our users. One of the use cases that will be used the most is client-side data querying such as filtering or sorting. We can also display graph and charts to help user gain insight easier.

4. Deliverables

- Design documents or proposal

  Before implementing this project, I will create a design document or proposal according to the organization format. This document or proposal will be reviewed and approved by the mentor and reviewer.
- New KubeArmor Coach service

  There will be a new service called KubeArmor Coach which will be used as the main place where all the complex computation and algorithm is implemented for all the features in this project. This service will also be responsible for storing or getting the data from the database.
- New KubeArmor Coach database

  Database that will be used to store observability data.
- New KubeArmor Cronjob

  Cronjob that will be used to remove old data from the database to ensure there is enough space all the time. This Cronjob then will be extended to implement smart tiering.
- KubeArmor DaemonSet modification

  KubeArmor DaemonSet needs to implement the GRPC method that will be called by KubeArmor Coach to get the observability data from each node.
- KubeArmor CLI modification

  There are few changes that need to be done on KubeArmor CLI such as implementing new commands, calling the GRPC service, and also presenting data in the most user-friendly way as text-based user interface.
- Unit testing for all features implemented in this project

  All features that are implemented in this project will get their corresponding unit test to ensure all code meets the quality standard before it’s deployed.
- Documentation

  To make sure that all users can utilize these new features effectively, there needs to be documentation. Documentation will help us ensure that everyone can have the same understanding on how to use or even develop this feature.

Key Dates	Task
Community Bonding Period	
Preparation May 20 - May 27	- Make the final draft of the design and project proposal according to the organization format of proposal documents
Preparation May 30 - Jun 3	- Ask for feedback and review of the design and project proposal. - Address the feedback that is given - Present the design and project proposal on KubeArmor community meeting
Preparation Jun 6 - Jun 10	- Finalize the design and project proposal documents - Start the setup and project initialisation for the new KubeArmor Coach repository
Coding Period Begins	
Week 1 Jun 13 - Jun 17	- Implement the protobuf definition and handler on KubeArmor Coach service - Implement basic database queries
Week 2 Jun 20 - Jun 24	- Implement the protobuf definition and handler on KubeArmor DaemonSet - Implement more complex queries used for filters.
Week 3 Jun 27 - Jul 1	- Integrate with KubeArmor DaemonSet to send data to KubeArmor Coach service and store it in the database.
Week 4 Jul 4 - Jul 8	- Integrate with KubeArmor DaemonSet to send data to KubeArmor Coach service and store it in the database. - Implement KubeArmor CLI *observe* command extension
Week 5	- Integrate with KubeArmor DaemonSet to send data to

Jul 11 - Jul 15	KubeArmor Coach service and store it in the database. - Integrate KubeArmor CLI to use the gRPC call
Week 6 Jul 18 - Jul 22	- Comprehensive end to end testing - Do some fixing if there are any bugs
First Evaluation	
Week 7 Jul 25 - Jul 29	- Implement KubeArmor Cronjob to remove old records.
Week 8 Aug 1 - Aug 5	- Extend KubeArmor Cronjob to implement smart tiering
Week 9 Aug 8 - Aug 12	- Extend KubeArmor Cronjob to implement smart tiering
Week 10 Aug 15 - Aug 19	- Implement the TUI output improvement KubeArmor CLI
Week 11 Aug 22 - Aug 26	- Implement the TUI output improvement KubeArmor CLI - Address code review feedback - Do more comprehensive testing
Week 12 Aug 29 - Sep 2	- Comprehensive end to end testing on all features - Do some fixing if there are any bugs
Week 13 Sep 5 - Sep 9	- Create a documentation regarding all the features that is implemented
Final Evaluation	

5. Notes

- I will be available to work full time (a minimum of 40 hours per week) during the Google Summer of Code period.
- Try to maintain a Trello board to track down the progress and all task backlog of this project.
- Try to maintain a google doc with daily updates to the work done for the day.
- Communicate and contacting with the mentor daily to keep them in the loop of how the work is progressing
- Participate in KubeArmor community meeting
- Maintain a github meta-tracker repository containing the following to ensure all the data regarding this GSoC project on one place:
    - A notes folder containing all notes that I took while getting to understand the task better
    - A link to the google doc with daily update
    - Link of issues that I am currently working on/have closed
    - List of pull requests that I have opened for this project

6. About Me

6.1. Personal Information

Name:	Nathaniel Jason
Email:	nthnieljson@gmail.com
Github:	@nthnieljson
LinkedIn:	https://www.linkedin.com/in/nathanieljason/
University:	Bandung Institute of Technology, Indonesia
Timezone:	GMT +7 (Jakarta)

6.2. Pre-GSoC

I have started contributing to KubeArmor before GSoC. Here are the list of PR that I have done.

- https://github.com/kubearmor/KubeArmor/pull/672
- https://github.com/kubearmor/kubearmor-client/pull/62

6.3. Post-GSoC

After GSoC I am planning to continue contributing at KubeArmor and really ambitious to later become a maintainer on KubeArmor.

6.4. Why Me

Introduction

I am Nathaniel Jason, a third-year computer science student at Bandung Institute of Technology, Indonesia. Since my first year in college, I started to find excitement and passion in learning full-stack web development. But recently, I discovered another part of software engineering that I find interesting, which is infrastructure and devops. For the past few months, I have been exploring a lot of microservices, Kubernetes, Docker, ArgoCD, Istio, and lots of other things.

Throughout internships and hackathons, I have been able to channel my passion for software engineering. I have interned in companies that vary in size, industry, and also positions covering full-stack software engineers and backend engineers. Aside from that, I have won some hackathons on a national and international level through solving problems providing a tech solution that brings impact.

Why am I interested in this mentorship opportunity?

One of the things that makes me interested in this mentorship opportunity is that we are being guided to contribute to an open-source project. I am optimistic that contributing to open-source projects will increase my software engineering skills and knowledge. What makes me more interested in this mentorship opportunity is that the open-source project that I'll be doing is Kubernetes related, which I find interesting and have been exploring for the past months. The other reason that I am interested in this mentorship opportunity is that I get the privilege to be guided and evaluated by mentors that are highly skilled in the field. I find Rahul Jadhav, one of the mentors, impressive in both technical and soft skills seeing from his experiences, therefore I really want to learn how to be a leader and technically skilled person at the same time like him. I also want to get to know and learn how to become a contributor on an open-source project from the other mentor, Barun Acharya. I am certain that acquiring knowledge from the mentors will be a great learning experience. The last reason why I am interested in this mentorship opportunity is that this will give me wide exposure to open-source projects and be an excellent first step for me to start my journey on open-source project contributions.

Through this mentorship experience, I am hoping to increase my software engineering skills and network, which I believe can be achieved by learning from mentors that are provided and contributing to open-source projects. Specifically, I want to increase my skills in Golang, Infrastructure, Kubernetes, and problem-solving. I am sure that the mentorship program in KubeArmor will be a good fit for me to improve those skills. I also want to learn more about KubeArmor and Kubernetes security since I realized that my current company, Gojek, hasn't leveraged Kubernetes security features the way it is supposed to. I want to implement the knowledge that I got from the KubeArmor open-source project into my current company.

Software engineering skills and networking abilities are key aspects to becoming a successful software engineer. Quoting one of my mentors that I have met, there are two key aspects that are needed to become a successful software engineer, that is you have to be really good at software engineering and you have to make sure people know that you are really good at it. I am sure that networking abilities help us to make people know our values and skills. I am also hoping that this mentorship experience will provide me guidance on contributing to open-source projects which will be a great foundation for me to become an open-source contributor in the future.

Past Work Experiences

There are technical and soft skills that I can apply to this mentorship program. On my last internship experience, I interned at Gojek, Southeast Asia's leading on-demand platform. Specifically, I interned at GoPay, Gojek's e-wallet that is supporting billions of dollars transactions, in the infrastructure and devops team. In this internship experience, I learned about Kubernetes, Istio, ArgoCD, microservice architecture, and many other things that are infrastructure-related. This means that I already have exposure to cloud and infrastructure context and domain knowledge. Onboarding to this project will be smooth since I am already exposed to the context and domain knowledge. I also have been using Golang, the programming language that is used in this project, for about 1 year, making it quick for me to start contributing to the codebase. I am also confident in my soft skills, especially in communicating and working together in a professional environment, which can be seen from my internship experience. I have about 1 year of internship experience in companies that vary in industry and size.

What makes me unique as an applicant?

One thing that makes me unique as an applicant is that I really enjoy learning new things and constantly seeking out new learning opportunities. This trait is represented by my various internship experiences and awards from competitions on a national to an international scale. I find it exciting to learn new tech stacks and frameworks which can be seen from the last few months, I have been exploring a lot of microservices, Kubernetes, Docker, ArgoCD, Istio, and many other things. If I have to learn new tech stacks or frameworks in the mentorship program, it won't be an obstacle for me, but a great learning opportunity. Another aspect that makes me unique as an applicant is that I came from a country that is underrepresented in open-source projects environments, which is Indonesia. According to https://k8s.devstats.cncf.io/, in 2021, Indonesia's contribution in open-source is only around 0.084% despite being one of the countries with the largest population. As a university student in Indonesia, I find that my fellow university students are not yet aware of open-source project contributions. For me, an open-source project is a great opportunity for us to contribute, learn and expand our network. I am confident that through this mentorship opportunity, I can advocate and introduce open-source project contributions to more students in Indonesia resulting in the expansion of the open-source environment.

7. References

- https://kubearmor.io/
- https://github.com/cncf/mentoring/blob/main/summerofcode/2022.md#observability-and-policy-discovery-helper-tool
- https://github.com/kubearmor/KubeArmor/issues/613
- https://blog.netwrix.com/2021/03/03/nist-800-53/#:~:text=NIST%20800%2D53%20is%20a,Information%20Technology%20Laboratory%20(ITL).
- https://csrc.nist.gov/csrc/media/publications/sp/800-53/rev-5/draft/documents/sp800-53r5-draft.pdf
- https://stackoverflow.com/questions/441001/possible-to-do-a-mysql-foreign-key-to-one-of-two-possible-tables
- https://blog.flant.com/announcing-elasticsearch-extractor-open-source-tool/
- https://logz.io/blog/smart-tiering/
- https://cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage