KubeArmor Dashboards

Abstract

In order to enhance user accessibility to monitor the behavior and state of applications and Kubearmor within Kubernetes clusters, we propose the implementation of dashboards integrated into the graphical user interface (GUI). While currently users can view Kubearmor application behavior and state through its CLI tool, we aim to provide a more user-friendly approach through GUI dashboards.

Specifically, we plan to integrate two types of dashboards: one for displaying Kubearmor state and behavior, and another for monitoring application behavior using Kibana or Grafana.

For the Kubearmor component, we intend to utilize the Kubernetes dashboard plugin functionality. This feature enables the integration of custom plugins into the Kubernetes dashboard, allowing users to visualize and manage Kubearmor state and behavior directly within the GUI. These plugins will be sourced from a configuration map, where their code will be parsed and displayed as components within the dashboard interface.

As for monitoring application behavior, we will explore the development of plugins for Kibana or Grafana. These plugins will be designed to interpret application logs and present relevant behavioral insights within the dashboard interface. Similar to the existing alert plugin integration with the Elastic Stack, these plugins will enhance the monitoring capabilities of the GUI by providing detailed insights into application behavior.

Terminologies

AOT - Ahead of time compilation
CRD - Custom resource definition

KubeArmor dashboard plugin

Creating Kubernetes dashboard plugin

Steps to create kubernetes dashboard plugin

1. compile your plugin source with Ahead of time compilation(AOT) with the following webpack config. webpack config
2. Load the compiled plugin source code into a config map eg) `kubectl create configmap karmorplugin -from-file="./dist/karmorplugin.js".`

3. Register a customCRD called Plugin by using this yaml

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/hack
/test-resources/plugin-crd.yml
```

4. Create an instance of that CRD and add the config map reference and filename and dependencies.
   example - plugin.yml

Unfortunately, the plugin feature isn't working as expected. To figure out why, we'll need to dive into how it's supposed to function. You can skip this part if you prefer and jump to the difficulties section.

Kubernetes Dashboard Plugin Working

Backend

Endpoints exposed for plugins

```
func (h *Handler) Install(ws *restful.WebService) {
    ws.Route(
       ws.GET("/plugin/config").
             To(h.handleConfig),
    )

    ws.Route(
       ws.GET("/plugin/{namespace}").
             To(h.handlePluginList).
             Writes(PluginList{}))

    ws.Route(
       ws.GET("/plugin/{namespace}/{pluginName}").
             To(h.servePluginSource))
}
```
reference

The plugin is served by this servePluginSource function which sends plugin source code as response on successful retrieval of source code from the configmap.

The below function does the source code retrieval from the configmap by using the namespace name and plugin name which is called by the servePluginSource function.

```
func GetPluginSource(ctx context.Context, client
pluginclientset.Interface, k8sClient kubernetes.Interface, ns
string, name string) ([]byte, error)
```
reference

Frontend

The plugin list is first retrieved in the plugins section which is loaded by the ‘kd-plugin-list’ Component.

```
  @Input() endpoint = EndpointManager.resource(Resource.plugin,
true).list();

  getResourceObservable(params?: HttpParams): Observable<PluginList> {
      return this.plugin_.get(this.endpoint, undefined, undefined,
params);
  }
```

The Endpoint for the request is received from the Endpoint Manager service which has three main methods which returns the endpoint url for retrieving the resources those are:

```
  list(): string {
      return `${baseHref}/${this.resource_}${this.namespaced_ ?
'/:namespace' : ''}`;
  }

  detail(): string {
      return `${baseHref}/${this.resource_}${this.namespaced_ ?
'/:namespace' : ''}/:name`;
  }

  child(resourceName: string, relatedResource: Resource,
resourceNamespace?: string): string {
      if (!resourceNamespace) {
      resourceNamespace = ':namespace';
      }

      return `${baseHref}/${this.resource_}${
      this.namespaced_ ? `/${resourceNamespace}` : ''
      }/${resourceName}/${relatedResource}`;
  }
```

In Case of plugins list the list method is used which returns the endpoint for getting the list of plugins as we saw in the backend part the endpoint for getting the plugin list have the same format. This plugin list component is loaded by the components in the plugins directory in the frontend part of kubernetes dashboard.

Routes:

```
export const PLUGIN_LIST_ROUTE: Route = {
  path: '',
  component: PluginListComponent,
  data: {
      breadcrumb: BREADCRUMBS.Plugins,
  },
};

export const PLUGIN_DETAIL_ROUTE: Route = {
  path: ':pluginNamespace/:pluginName',
  component: PluginDetailComponent,
  data: {
      breadcrumb: '{{ pluginName }}',
      parent: PLUGIN_LIST_ROUTE,
  },
};
```

The plugin list component is loaded by default on the /plugin route. When we click the plugin it routes to /plugin/pluginNamespace/PluginName. These Namespace and Pluginname are the require data for the retrieval of plugin source code as we saw in the backend part so by using this data we could get the source code and load them into the dashboard. So when the route is changed the pluginDetailcomponent is loaded which passes the pluginname from the path and send to its child component “kd-plugin-holder”.

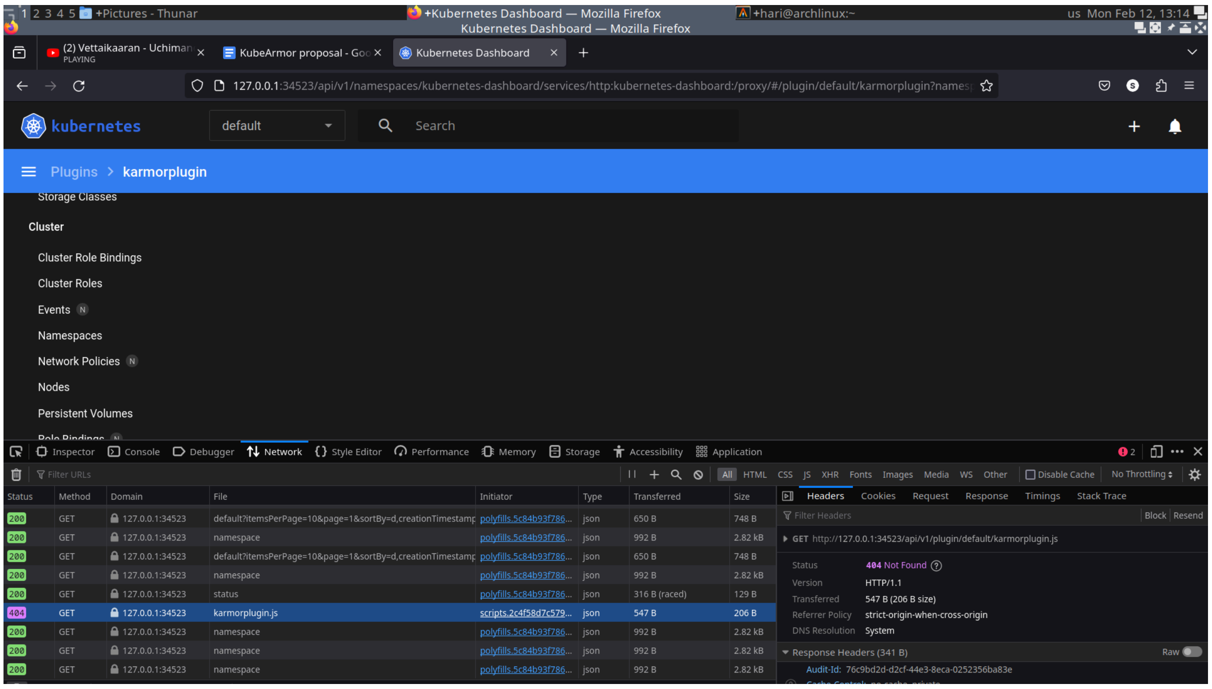
This is the main component for loading the plugin. This component class has a method called loadPlugin which is responsible for loading the plugin. It uses the pluginLoader service to get the plugin source and the code is changed to module. Once the plugin module is loaded, it creates a module reference and attempts to resolve the entry component defined in the module. If successful, it creates the component dynamically within the view container referenced by `vcRef`. If there's an error during this process, it sets `entryError` to true. The component conditionally renders the UI if the entryError is true Then it will be displayed a no entry component else plugin will be loaded

The difficulties while building and loading the plugin.

The given docs for dashboard plugins was not sufficient for building the plugin as the code repo was outdated and had a lot of dependency conflicts. So in order to compile we need to compile a plugin using a custom builder as mentioned in the docs. The custom builder basically modifies the webpack config and imports the ngfactory modules to the entry point which is our main.ts and compiles the code which can be effectively achieved by having a webpack config and building using webpack. So I Am able to build a simple plugin with a single element

```
<main>hello world</main>
```

I compiled the plugin and followed the aforementioned steps for creating a kubernetes dashboard plugin but the plugin was not loaded into the UI. I debugged and found that the plugin resource was not found at the requested endpoint



+Kubernetes Dashboard — Mozilla Firefox
127.0.0.1:34523/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/#/plugin/default/karmorplugin?names
kubernetes
default
Search
Plugins > karmorplugin
Storage Classes
Cluster
Cluster Role Bindings
Cluster Roles
Events
Namespaces
Network Policies
Nodes
Persistent Volumes
Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility Application

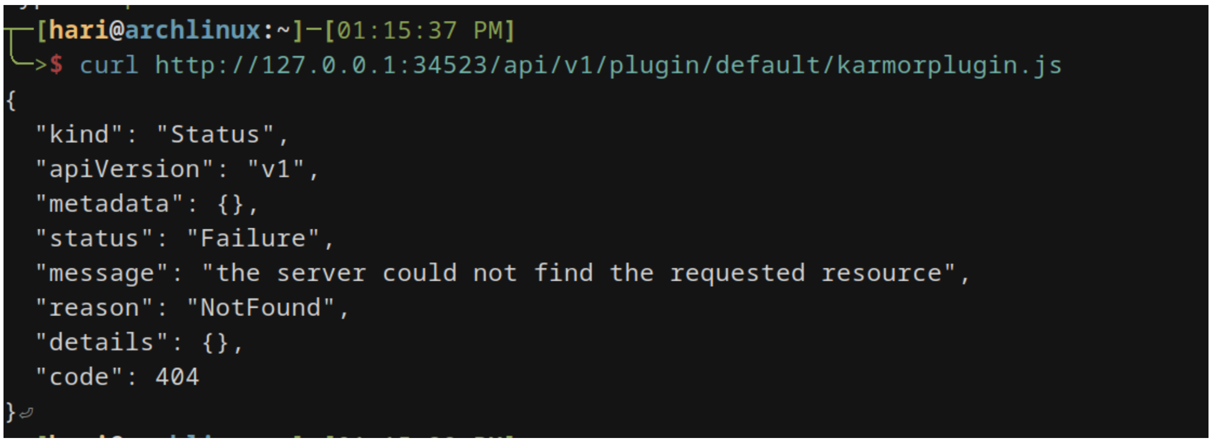
Status	Method	Domain	File	Initiator	Type	Transferred	Size
200	GET	127.0.0.1:34523	default?itemsPerPage=10&page=1&sortBy=d,creationTimestamp	polyfills.5c84b93f786...	json	650 B	748 B
200	GET	127.0.0.1:34523	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:34523	default?itemsPerPage=10&page=1&sortBy=d,creationTimestamp	polyfills.5c84b93f786...	json	650 B	748 B
200	GET	127.0.0.1:34523	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:34523	status	polyfills.5c84b93f786...	json	316 B (raced)	129 B
404	GET	127.0.0.1:34523	karmorplugin.js	scripts.2c4f58d7c579...	json	547 B	206 B
200	GET	127.0.0.1:34523	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:34523	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:34523	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB

GET http://127.0.0.1:34523/api/v1/plugin/default/karmorplugin.js

Status	404 Not Found
Version	HTTP/1.1
Transferred	547 B (206 B size)
Referrer Policy	strict-origin-when-cross-origin
DNS Resolution	System

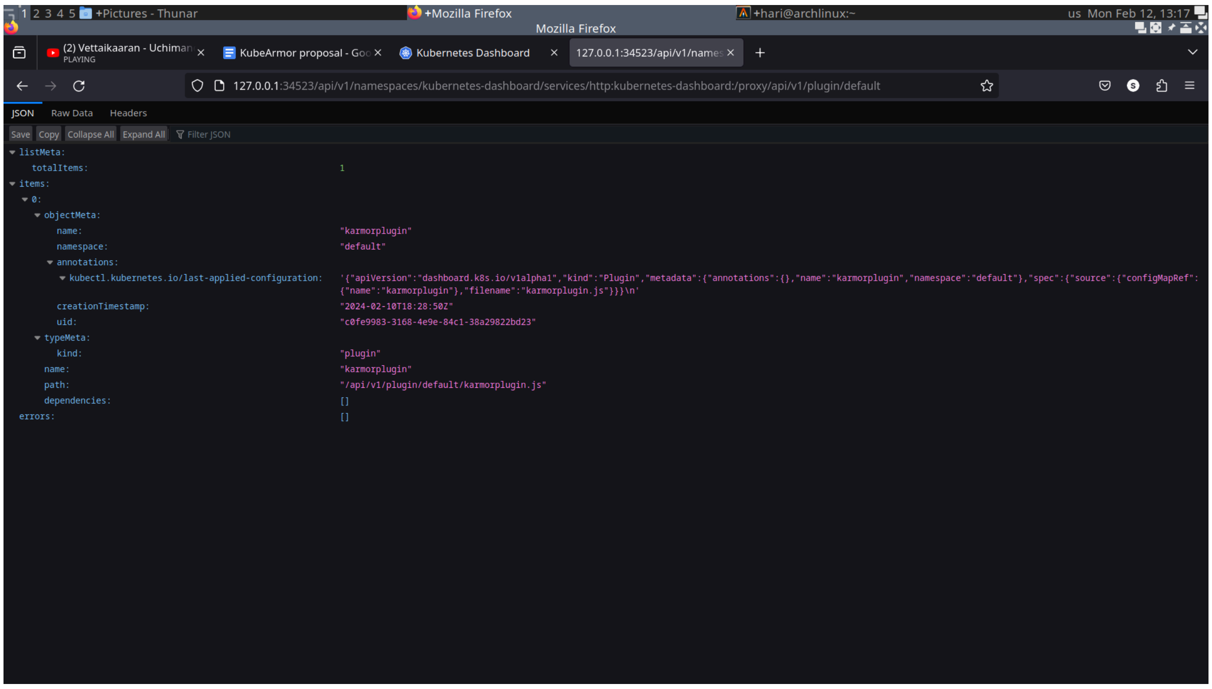
Response Headers (341 B)

Audit-Id: 76c9bd2d-d2cf-44e3-8eca-0252356ba83e



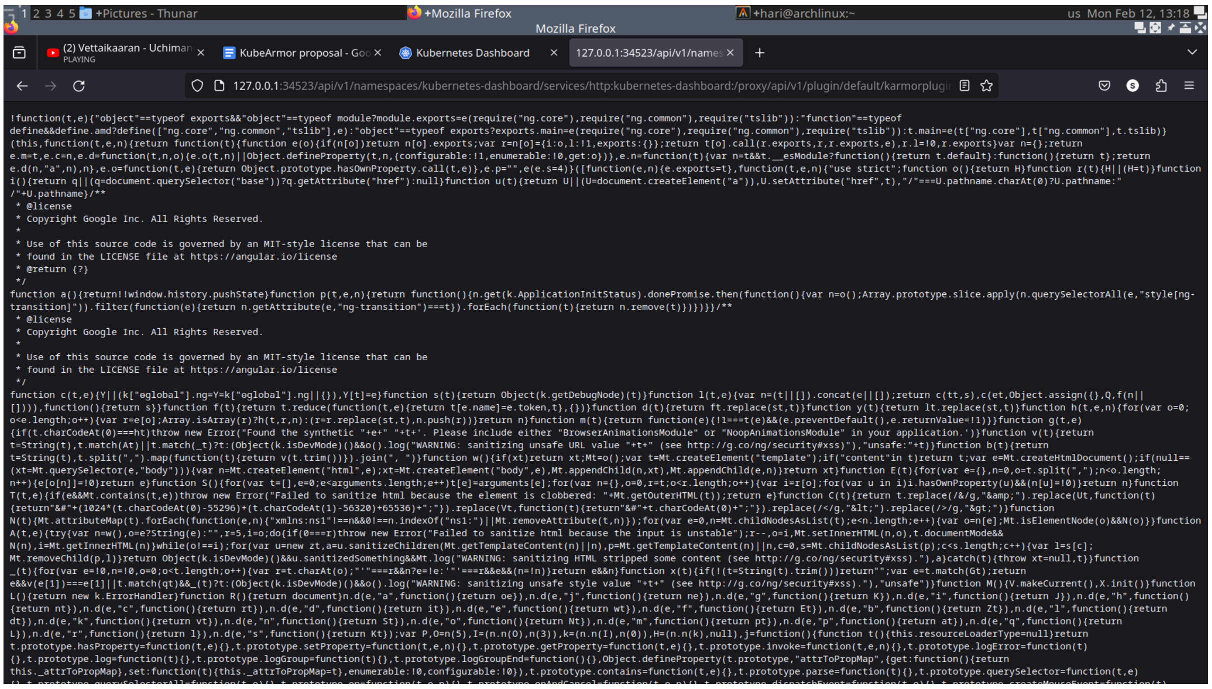
```
┌─[hari@archlinux:~]─[01:15:37 PM]
└─>$ curl http://127.0.0.1:34523/api/v1/plugin/default/karmorplugin.js
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "the server could not find the requested resource",
  "reason": "NotFound",
  "details": {},
  "code": 404
}
```

Upon observing the images above, it's evident that the status code returned is 404, indicating that the requested resource was not found at the specified endpoint. Consequently, I proceeded to verify the endpoint for the list of plugins.



```
127.0.0.1:34523/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/api/v1/plugin/default

▾ listMeta:
    totalItems:                                   1
▾ items:
  ▾ 0:
    ▾ objectMeta:
        name:                                     "karmorplugin"
        namespace:                                "default"
      ▾ annotations:
        ▾ kubectl.kubernetes.io/last-applied-configuration:   '{"apiVersion":"dashboard.k8s.io/v1alpha1","kind":"Plugin","metadata":{"annotations":{},"name":"karmorplugin","namespace":"default"},"spec":{"source":{"configMapRef":
                                                  {"name":"karmorplugin"},"filename":"karmorplugin.js"}}}\n'
        creationTimestamp:                        "2024-02-10T18:28:50Z"
        uid:                                      "c0fe9983-3168-4e9e-84c1-38a29822bd23"
    ▾ typeMeta:
        kind:                                     "plugin"
      name:                                       "karmorplugin"
      path:                                       "/api/v1/plugin/default/karmorplugin.js"
      dependencies:                               []
  errors:                                         []
```

The plugin list endpoint is functioning properly and successfully retrieves the list of plugins, as depicted in the above image. In the backend section, we observed that the endpoint for retrieving a plugin follows the format "/plugin/namespace/pluginname". Consequently, I adjusted the endpoint of the plugins list by appending the plugin name at the end. This modification proved successful, as the source code was loaded in the browser.


1 2 3 4 5 +Pictures - Thunar
+Mozilla Firefox
+hari@archlinux:~
us Mon Feb 12, 13:18
Mozilla Firefox
(2) Vettaikaaran - Uchiman
PLAYING
KubeArmor proposal - Goo
Kubernetes Dashboard
127.0.0.1:34523/api/v1/names
127.0.0.1:34523/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/api/v1/plugin/default/karmorplugi

```
!function(t,e){"object"==typeof exports&&"object"==typeof module?module.exports=e(require("ng.core"),require("ng.common"),require("tslib")):"function"==typeof
define&&define.amd?define(["ng.core","ng.common","tslib"],e):"object"==typeof exports?exports.main=e(require("ng.core"),require("ng.common"),require("tslib")):t.main=e(t["ng.core"],t["ng.common"],t.tslib)}
(this,function(t,e,n){return function(t){function e(o){if(n[o])return n[o].exports;var r=n[o]={i:o,l:!1,exports:{}};return t[o].call(r.exports,r,r.exports,e),r.l=!0,r.exports}var n={};return
e.m=t,e.c=n,e.d=function(t,n,o){e.o(t,n)||Object.defineProperty(t,n,{configurable:!1,enumerable:!0,get:o})},e.n=function(t){var n=t&&t.__esModule?function(){return t.default}:function(){return t};return
e.d(n,"a",n),n},e.o=function(t,e){return Object.prototype.hasOwnProperty.call(t,e)},e.p="",e(e.s=4)}([function(e,n){e.exports=t},function(t,e,n){"use strict";function o(){return H}function r(t){H||(H=t)}function
i(){return q||(q=document.querySelector("base"))?q.getAttribute("href"):null}function u(t){return U||(U=document.createElement("a")),U.setAttribute("href",t),"/"===U.pathname.charAt(0)?U.pathname:"
/"+U.pathname}/**
 * @license
 * Copyright Google Inc. All Rights Reserved.
 *
 * Use of this source code is governed by an MIT-style license that can be
 * found in the LICENSE file at https://angular.io/license
 * @return {?}
 */
function a(){return!!window.history.pushState}function p(t,e,n){return function(){n.get(k.ApplicationInitStatus).donePromise.then(function(){var n=o();Array.prototype.slice.apply(n.querySelectorAll(e,"style[ng-
transition]")).filter(function(e){return n.getAttribute(e,"ng-transition")===t}).forEach(function(t){return n.remove(t)})})}}/**
 * @license
 * Copyright Google Inc. All Rights Reserved.
 *
 * Use of this source code is governed by an MIT-style license that can be
 * found in the LICENSE file at https://angular.io/license
 */
function c(t,e){Y||(k["ɵglobal"].ng=Y=k["ɵglobal"].ng||{}),Y[t]=e}function s(t){return Object(k.getDebugNode)(t)}function l(t,e){var n=(t||[]).concat(e||[]);return c(tt,s),c(et,Object.assign({},Q,f(n||
[]))),function(){return s}}function f(t){return t.reduce(function(t,e){return t[e.name]=e.token,t},{})}function d(t){return ft.replace(st,t)}function y(t){return lt.replace(st,t)}function h(t,e,n){for(var o=0;
o<e.length;o++){var r=e[o];Array.isArray(r)?h(t,r,n):(r=r.replace(st,t),n.push(r))}return n}function m(t){return function(e){!1===t(e)&&(e.preventDefault(),e.returnValue=!1)}}function g(t,e)
{if(t.charCodeAt(0)===ht)throw new Error("Found the synthetic "+e+" "+t+'. Please include either "BrowserAnimationsModule" or "NoopAnimationsModule" in your application.')}function v(t){return
t=String(t),t.match(At)||t.match(_t)?t:(Object(k.isDevMode)()&&o().log("WARNING: sanitizing unsafe URL value "+t+" (see http://g.co/ng/security#xss)"),"unsafe:"+t)}function b(t){return
t=String(t),t.split(",").map(function(t){return v(t.trim())}).join(", ")}function w(){if(xt)return xt;Mt=o();var t=Mt.createElement("template");if("content"in t)return t;var e=Mt.createHtmlDocument();if(null==
(xt=Mt.querySelector(e,"body"))){var n=Mt.createElement("html",e);xt=Mt.createElement("body",e),Mt.appendChild(n,xt),Mt.appendChild(e,n)}return xt}function E(t){for(var e={},n=0,o=t.split(",");n<o.length;
n++){e[o[n]]=!0}return e}function S(){for(var t=[],e=0;e<arguments.length;e++)t[e]=arguments[e];for(var n={},o=0,r=t;o<r.length;o++){var i=r[o];for(var u in i)i.hasOwnProperty(u)&&(n[u]=!0)}return n}function
T(t,e){if(e&&Mt.contains(t,e))throw new Error("Failed to sanitize html because the element is clobbered: "+Mt.getOuterHTML(t));return e}function C(t){return t.replace(/&/g,"&amp;").replace(Ut,function(t)
{return"&#"+(1024*(t.charCodeAt(0)-55296)+(t.charCodeAt(1)-56320)+65536)+";"}).replace(Vt,function(t){return"&#"+t.charCodeAt(0)+";"}).replace(/</g,"&lt;").replace(/>/g,"&gt;")}function
N(t){Mt.attributeMap(t).forEach(function(e,n){"xmlns:ns1"!==n&&0!==n.indexOf("ns1:")||Mt.removeAttribute(t,n)});for(var e=0,n=Mt.childNodesAsList(t);e<n.length;e++){var o=n[e];Mt.isElementNode(o)&&N(o)}}function
A(t,e){try{var n=w(),o=e?String(e):"",r=5,i=o;do{if(0===r)throw new Error("Failed to sanitize html because the input is unstable");r--,o=i,Mt.setInnerHTML(n,o),t.documentMode&&
N(n),i=Mt.getInnerHTML(n)}while(o!==i);for(var u=new zt,a=u.sanitizeChildren(Mt.getTemplateContent(n)||n),p=Mt.getTemplateContent(n)||n,c=0,s=Mt.childNodesAsList(p);c<s.length;c++){var l=s[c];
Mt.removeChild(p,l)}return Object(k.isDevMode)()&&u.sanitizedSomething&&Mt.log("WARNING: sanitizing HTML stripped some content (see http://g.co/ng/security#xss)."),a}catch(t){throw xt=null,t}}function
_(t){for(var e=!0,n=!0,o=0;o<t.length;o++){var r=t.charAt(o);"'"===r&&n?e=!e:'"'===r&&e&&(n=!n)}return e&&n}function x(t){if(!(t=String(t).trim()))return"";var e=t.match(Gt);return
e&&v(e[1])===e[1]||t.match(qt)&&_(t)?t:(Object(k.isDevMode)()&&o().log("WARNING: sanitizing unsafe style value "+t+" (see http://g.co/ng/security#xss)."),"unsafe")}function M(){V.makeCurrent(),X.init()}function
L(){return new k.ErrorHandler}function R(){return document}n.d(e,"a",function(){return oe}),n.d(e,"j",function(){return ne}),n.d(e,"g",function(){return K}),n.d(e,"i",function(){return J}),n.d(e,"h",function()
{return nt}),n.d(e,"c",function(){return rt}),n.d(e,"d",function(){return it}),n.d(e,"e",function(){return wt}),n.d(e,"f",function(){return Et}),n.d(e,"b",function(){return Zt}),n.d(e,"l",function(){return
dt}),n.d(e,"k",function(){return vt}),n.d(e,"n",function(){return St}),n.d(e,"o",function(){return Nt}),n.d(e,"m",function(){return pt}),n.d(e,"p",function(){return at}),n.d(e,"q",function(){return
L}),n.d(e,"r",function(){return l}),n.d(e,"s",function(){return Kt});var P,O=n(5),I=(n.n(O),n(3)),k=(n.n(I),n(0)),H=(n.n(k),null),j=function(){function t(){this.resourceLoaderType=null}return
t.prototype.hasProperty=function(t,e){},t.prototype.setProperty=function(t,e,n){},t.prototype.getProperty=function(t,e){},t.prototype.invoke=function(t,e,n){},t.prototype.logError=function(t)
{},t.prototype.log=function(t){},t.prototype.logGroup=function(t){},t.prototype.logGroupEnd=function(){},Object.defineProperty(t.prototype,"attrToPropMap",{get:function(){return
this._attrToPropMap},set:function(t){this._attrToPropMap=t},enumerable:!0,configurable:!0}),t.prototype.contains=function(t,e){},t.prototype.parse=function(t){},t.prototype.querySelector=function(t,e)
```

So it is clear that the api's are working perfectly so we need to change the endpoint in the frontend and load in the plugin.

The current endpoint for loading the plugin:
"http://127.0.0.1:port/api/v1/plugin/default/karmorplugin.js"

The actual endpoint for getting the source code of the plugin:
"http://127.0.0.1:port/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/api/v1/plugin/default/karmorplugin"

Hence ,this issue needs to be solved for loading and testing the basic version of the kubearmor plugin.

Overview of possible approaches

1. Kubernetes native API

The KubeArmor state data can be obtained from the native kubernetes API since we are compiling and building the plugin we need to have a static endpoint for obtaining the resources in case of this native api which is accessible through `kubectl proxy` command which has the default port 8001.

if the port of the proxy changes then the endpoint for accessing the resources will also change so the api request from the plugins will fail.

This approach will also not have a consistent and modular approach as we need to get the resources using custom urls and query parameters.

2. Kubernetes dashboard API

Kubernetes dashboard provides the necessary API for obtaining the kubernetes resources and it is a preferred approach compared to other approaches. The endpoint of kubernetes dashboard is consistent and authenticated so it eliminates concerns regarding port changes when running kubectl proxy. The kubernetes dashboard already provides services for accessing resources using the endpoint so we don’t need to worry about managing the endpoint as it is modularized and existing as services. The endpoint can be easily integrated with a plugin and can be compiled with it.

The only major con of this approach would be the changes in the backend of the dashboard but the endpoints are stable for a longer time so I think it wont affect the plugin in the future

3. Custom server for KubeArmor plugin

This is the first approach which came to my mind for obtaining the kubernetes resources. Since we already have relay server running we can add an additional server to it for sending the requested data.

This idea will be good if we had multiple clients and needs kubearmor state data quite often but in our case we have only 2 client one is karmor cli which show kubearmor state by running karmor probe command and the other client is kubearmor plugin and we need the data quite rarely to view the state so this approach is least preferred as it requires to run a dedicated server for a small use case so it is quite expensive.

Final approach

From the overview of approaches it is clear that using kubernetes dashboard API would be a better approach for obtaining the kubernetes resources.

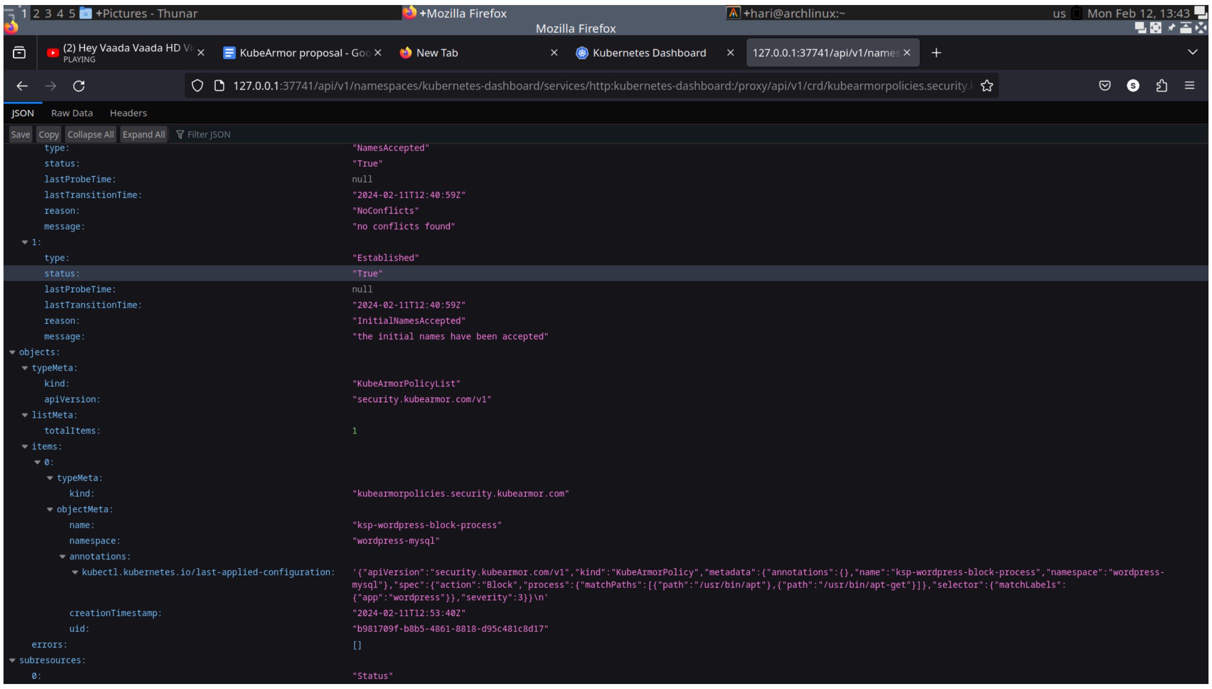
Implementation of the Kube Armor Plugin

KubeArmor Policies and KubeArmor Host policies

Endpoints for CRDs

```
     apiV1Ws.Route(
      apiV1Ws.GET("/crd").

To(apiHandler.handleGetCustomResourceDefinitionList).
           Writes(types.CustomResourceDefinitionList{}))

    apiV1Ws.Route(
      apiV1Ws.GET("/crd/{crd}").

To(apiHandler.handleGetCustomResourceDefinitionDetail).
Writes(types.CustomResourceDefinitionDetail{}))

    apiV1Ws.Route(
      apiV1Ws.GET("/crd/{namespace}/{crd}/object").

To(apiHandler.handleGetCustomResourceObjectList).
          Writes(types.CustomResourceObjectList{}))
```

We can use these endpoints to get the KubeArmor policies and Kubearmor Host policies and other data for displaying the KubeArmor state. As shown in the image below we are able to get the kubearmor policies.



1 2 3 4 5 +Pictures - Thunar
+Mozilla Firefox
+hari@archlinux:~
us Mon Feb 12, 13:43
Mozilla Firefox
(2) Hey Vaada Vaada HD Vi
PLAYING
KubeArmor proposal - Goo
New Tab
Kubernetes Dashboard
127.0.0.1:37741/api/v1/names
127.0.0.1:37741/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/api/v1/crd/kubearmorpolicies.security.k
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
      type:                         "NamesAccepted"
      status:                       "True"
      lastProbeTime:                null
      lastTransitionTime:           "2024-02-11T12:40:59Z"
      reason:                       "NoConflicts"
      message:                      "no conflicts found"
  ▾ 1:
      type:                         "Established"
      status:                       "True"
      lastProbeTime:                null
      lastTransitionTime:           "2024-02-11T12:40:59Z"
      reason:                       "InitialNamesAccepted"
      message:                      "the initial names have been accepted"
▾ objects:
  ▾ typeMeta:
      kind:                         "KubeArmorPolicyList"
      apiVersion:                   "security.kubearmor.com/v1"
  ▾ listMeta:
      totalItems:                   1
  ▾ items:
    ▾ 0:
      ▾ typeMeta:
          kind:                     "kubearmorpolicies.security.kubearmor.com"
      ▾ objectMeta:
          name:                     "ksp-wordpress-block-process"
          namespace:                "wordpress-mysql"
        ▾ annotations:
          ▾ kubectl.kubernetes.io/last-applied-configuration: '{"apiVersion":"security.kubearmor.com/v1","kind":"KubeArmorPolicy","metadata":{"annotations":{},"name":"ksp-wordpress-block-process","namespace":"wordpress-
                                    mysql"},"spec":{"action":"Block","process":{"matchPaths":[{"path":"/usr/bin/apt"},{"path":"/usr/bin/apt-get"}]},"selector":{"matchLabels":
                                    {"app":"wordpress"}},"severity":3}}\n'
          creationTimestamp:        "2024-02-11T12:53:40Z"
          uid:                      "b981709f-b8b5-4861-8818-d95c481c8d17"
    errors:                         []
▾ subresources:
    0:                              "Status"
```

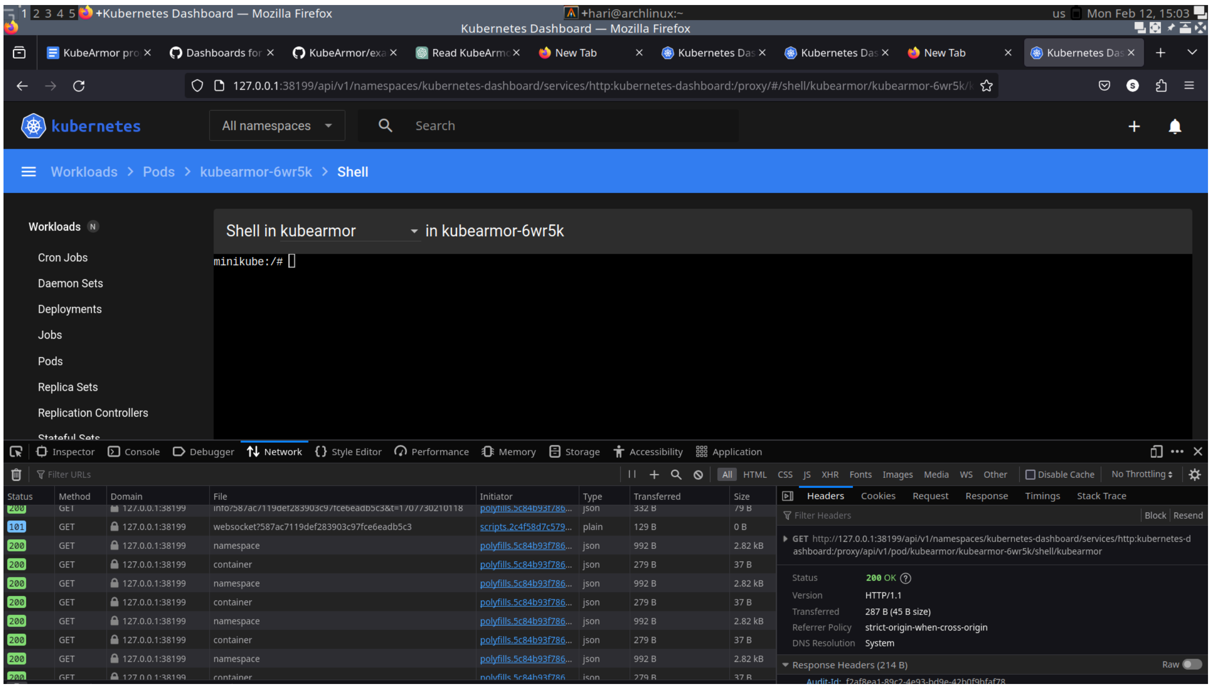
ProbeData

The probe data is stored within the "karmorProbeData.cfg" file located in the KubeArmor pod's "/tmp" directory. To access this data, the "cat" command needs to be executed, a task achievable through the Kubernetes dashboard API.

```
     apiV1Ws.Route(

apiV1Ws.GET("/pod/{namespace}/{pod}/shell/{container}").
          To(apiHandler.handleExecShell).
          Writes(TerminalResponse{}))
```

This endpoint creates a new terminal session in the requested pod which can be used to retrieve the kubearmor probe data. After obtaining the terminal session we can use web sockets to execute the required commands . Posture and Visibility can be collected from the probe data.

References:
[API]
[TerminalSession]



Shell in kubearmor in kubearmor-6wr5k

```
minikube:/#
```

Status	Method	Domain	File	Initiator	Type	Transferred	Size
200	GET	127.0.0.1:38199	info?587ac7119def283903c97fce6eadb5c3&t=1707730210118	polyfills.5c84b93f786...	json	332 B	79 B
101	GET	127.0.0.1:38199	websocket?587ac7119def283903c97fce6eadb5c3	scripts.2c4f58d7c579...	plain	129 B	0 B
200	GET	127.0.0.1:38199	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:38199	container	polyfills.5c84b93f786...	json	279 B	37 B
200	GET	127.0.0.1:38199	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:38199	container	polyfills.5c84b93f786...	json	279 B	37 B
200	GET	127.0.0.1:38199	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:38199	container	polyfills.5c84b93f786...	json	279 B	37 B
200	GET	127.0.0.1:38199	namespace	polyfills.5c84b93f786...	json	992 B	2.82 kB
200	GET	127.0.0.1:38199	container	polyfills.5c84b93f786...	json	279 B	37 B

GET http://127.0.0.1:38199/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/api/v1/pod/kubearmor/kubearmor-6wr5k/shell/kubearmor

Status	200 OK
Version	HTTP/1.1
Transferred	287 B (45 B size)
Referrer Policy	strict-origin-when-cross-origin
DNS Resolution	System

Response Headers (214 B)

Audit-Id: f2af8ea1-89c2-4e93-bd9e-42b0f9bfaf78

Probe data can be gathered for all nodes by utilizing the endpoint to retrieve information about all nodes and the pods running on each respective node.

```
     apiV1Ws.Route(
      apiV1Ws.GET("/node").
          To(apiHandler.handleGetNodeList).
          Writes(node.NodeList{}))
    apiV1Ws.Route(
      apiV1Ws.GET("/node/{name}").
          To(apiHandler.handleGetNodeDetail).
          Writes(node.NodeDetail{}))
    apiV1Ws.Route(
      apiV1Ws.GET("/node/{name}/event").
          To(apiHandler.handleGetNodeEvents).
          Writes(common.EventList{}))
    apiV1Ws.Route(
      apiV1Ws.GET("/node/{name}/pod").
          To(apiHandler.handleGetNodePods).
          Writes(pod.PodList{}))
```

We can check for the kubearmor pods by the labels and field selector for each node and collect their probe data.

```
     LabelSelector: "kubearmor-app=kubearmor",
      FieldSelector: "spec.nodeName=" + nodeName,
```

Using the probe Data we can get the posture data

```
type PostureData = {
  filePosture: string,
  capabilitiesPosture: string,
  networkPosture: string,
  visibility: string,

}
```

Armored Pods

The Armored pods within each namespace, along with their posture, visibility, and policies, can be obtained from the Posture and Policy data. The Annotated pods can be retrieved using following condition.

```
pod.Annotations["kubearmor-policy"] == "enabled"
```

```
interface PodInfo {
    PodName: string;
    Policy: string[];
}

interface NamespaceData {
    NsDefaultPosture: string;
    NsVisibilitystring: string;
   NsVisibility: Visibility;
    NsPodList: PodInfo[];
}

Interface Visibility{
File: boolean;
Capabilities : boolean;
Process: boolean;
Network: boolean;
}

interface ArmoredPodData {
    [key: string]: NamespaceData;
}

interface AnnotatedPodsResponse {
    Namespaces: ArmoredPodData;
}
```

We now have access to all the essential data needed to showcase the Kubearmor state. Just like the Karmor probe command, we can create tabular columns to present the KubearmorState data.

UI

For the user interface, we can utilize the angular/material library to construct the necessary components. This library can serve as a shared dependency, thereby minimizing the build size of the Kube Armor plugin.

Milestones for kubearmor plugin

1. Loading and displaying the sample plugin by changing the endpoint of the plugin source in the kubearmor dashboard frontend part. Ensuring the compilation is perfect and the entry point of the plugin is detectable.
2. Building services for obtaining the kubearmor state details and storing it.
3. Building UI components for displaying the kubearmor state.

Kibana/Grafana Dashboard

KubeArmor presently supports showcasing alerts in the Kibana dashboard via Elasticsearch. Although the existing relay server facilitates visualizing KubeArmor alerts, there's a requirement for a plugin to visualize cluster logs beyond violations.

By utilizing pre-existing log retrieval methods, I've enhanced the system to include bulk indexing of logs into Elasticsearch.

Plugin for viewing Logs in kibana dashboard

```
type ElasticsearchClient struct {
    kaClient   *server.LogClient
    esClient   *elasticsearch.Client
    cancel     context.CancelFunc
    bulkIndexer esutil.BulkIndexer
    ctx        context.Context
    alertCh    chan interface{}
    logCh      chan interface{}
}
```

```
     client.WgServer.Add(1)
    go func() {
      defer client.WgServer.Done()
      for client.Running {
            res, err := client.LogStream.Recv()
            if err != nil {
                  kg.Warnf("Failed to receive an log (%s)",
client.Server)
                  break
            }
            tel, _ := json.Marshal(res)
            fmt.Printf("%s\n", string(tel))
            ecl.logCh <- res
      }
    }()

    for i := 0; i < 5; i++ {
      go func() {
            for {
                  select {
                  case log := <-ecl.logCh:
                        ecl.bulkIndex(log, "log")
                        kg.Print("received log and indexed")
                  case <-ecl.ctx.Done():
                        close(ecl.logCh)
                        return
                    }
                }
            }()
    }
```

The `client.Logstream.Recv()` will receive the logs then the logs are passed to the logChannel which is received and bulkindexed with the index name “log”.

Now like alerts we have logs indexed with elastic search which can be viewed and queried using kibana and grafana.

Overview of possible approaches

1. **Creating Dashboards using json**
   This is the most common method of creating dashboards. The kibana dashboard can be built with the json files which is a straightforward approach and it is used in existing kubearmor dashboard for viewing alerts.
2. **Creating Dashboards programmatically using the dashboard goclient**
   This approaches involves utilizing the go-client libraries of the dashboards to build the dashboard in an automated way so that the user need not to apply the json and create a dashboard the sample dashboards will be ready for the user as soon as the user deploys the kibana and grafana dashboard. This approach has more control over the data compared to the json approach.
   In this approach we will be creating the go client in the kubearmor relay server.

Final Approach

Since the dashboards are mostly from the users side it will be easy and simple for users to have json approach they can modify and use the dashboards using json but in the go-client approach users cannot modify the existing default dashboard which will be created automatically from the relay server user cannot modify the existing default dashboard which might affect the user experience so the json based approach will be better in this case.

Milestones for kibana/grafana dashboard

1. Deploying elasticsearch and checking whether the logs index is available.
2. Integration with kibana and grafana by implementing the dashboard for viewing the logs.

Conclusion

The successful integration of the KubeArmor plugin into the Kubernetes dashboard hinges on the correct configuration of the endpoint, necessitating support from the Kubernetes dashboard team. In light of this, I am considering creating an issue and submitting a pull request to address this matter. Despite attempts to contact the project maintainers via the Slack channel, no response has been received thus far. An issue has already been created regarding the custom builder, with a request for guidance on referencing existing Kubernetes dashboard plugins.

Once the endpoint issue is resolved, attention will be directed towards rendering the plugin from its source code within the Kubernetes dashboard. Armed with a clear understanding of the problem, I intend to modify the endpoint and experiment with loading the plugin by building it as a custom container, thereby conducting a practical test of its functionality.

The recommended policies functionality could be added to the KubeArmor plugin and being able to add them to the pod would be a good feature which might have an impact on user experience.

Furthermore, implementing dashboards for application behavior is deemed relatively straightforward due to the abundance of examples available, making it easier compared to integrating the KubeArmor plugin.