PRESENTS

KubeEdge Security Audit

In collaboration with the KubeEdge project maintainers and The Open Source Technology Improvement Fund and commissioned by the Cloud Native Computing Foundation.

Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 11th July, 2022.

Executive summary

Results summarised
Formalisation of KubeEdge threat model
12 issues found
8 CVEs assigned
OSS-Fuzz and CIFuzz integration of KubeEdge
10 fuzzers developed
SLSA compliance review

This report outlines the security audit carried out by Ada Logics of KubeEdge May and June 2022. The audit was carried out by Ada Logics in collaboration with the KubeEdge maintainers and was commissioned by the CNCF and facilitated by OSTIF. Ada Logics dedicated a total of 25 days of effort for this audit.

The goal of this audit was to conduct a holistic security assessment of KubeEdge which included several high-level tasks. The audit had four three high-level tasks:

- Threat-model formalisation.
- Fuzzing integration.
- Manual code auditing.

In addition to these three main tasks, Ada Logics performed a SLSA compliance review, which is found in the Appendix of this report.

The threat-model was made for both KubeEdge users and security researchers interested in improving the security of KubeEdge. To assist KubeEdge users, the threat model identifies the areas where users need to ensure that their infrastructure around KubeEdge is secure and outlining the security implications in case users should deploy KubeEdge in insecure infrastructure. To assist security researchers, the threat model clarifies which security policies KubeEdge implements to indicate that any behaviour in KubeEdge that contradicts these intended policies can have security implications that should be brought to the attention of the KubeEdge team.

At the beginning of this audit KubeEdge had no exposure to fuzzing. In this audit, Ada Logics integrated KubeEdge into OSS-Fuzz, developed 10 fuzzers and set up these fuzzers to run in the CI for pull requests. Several issues were found by the fuzzers, and 2 CVEs were assigned. The KubeEdge team was fast to reproduce crashes and respond with patches.

The manual code auditing found several issues ranging from informational to moderate severity. 8 denial-of-service CVEs were assigned as a result of this part of the code audit. These were found in both EdgeCore and CloudCore.

Project information

Document History

Version	Date	Details
1.0	21/06/2022	First version shared with the KubeEdge team
1.1	05/07/2022	Final version of report delivered.

Contacts

Ada Logics

Name	**Title**	**Email address**
Adam Korczynski	Security Engineer	Adam@adalogics.com
David Korczynski	Security Researcher	David@adalogics.com

Open Source Technology Improvement Fund

Name	**Email**
Amir Montazery	Amir@ostif.org
Derek Zimmer	Derek@ostif.org

KubeEdge

Name	**Email**
Kevin Wang	wangzefeng@huawei.com
Fisher Xu	xufei40@huawei.com
Vincent Lin	linguohui1@huawei.com

Table of contents

Fuzzing

In this section we outline the fuzzing efforts involved in this holistic security audit. Prior to the audit, KubeEdge had done no fuzzing and, thus, Ada Logics developed an extensive fuzzing suite targeted at the KubeEdge project.

The first step in the fuzzing efforts was to integrate KubeEdge into OSS-Fuzz and CNCF-Fuzzing. The purpose of this step was to get an end-to-end working fuzzing integration, so that all KubeEdges fuzzers developed throughout the audit would be automatically run by OSS-Fuzz. The integration into OSS-Fuzz has the added benefit that each fuzzer will continuously run after completion of the audit. Following this first step, Ada Logics continued by adding more fuzzers to increase code coverage throughout the entire code base.

The fuzzers are built with the build script found here: https://github.com/cncf/cncf-fuzzing/blob/main/projects/kubeedge/build.sh. More fuzzers can be added to this script by building them by way of the `compile_go_fuzzer` binary.

All fuzzers were implemented in the go-fuzz fuzzing engine.

Fuzzers

This section outlines the fuzzers written as part of the security audit.

Overview

This table gives an overview of the fuzzers written during the security audit. “CNCF-Fuzzing repo” refers to this link: https://github.com/cncf/cncf-fuzzing/tree/main/projects/kubeedge

Fuzzer Name	Target package	Location
`FuzzLaneReadMessage`	`github.com/kubeedge/viaduct/pkg/lane`	CNCF-Fuzzing repo
`FuzzdealTwinActions`	`github.com/kubeedge/kubeedge/edge/pkg/devicetwin/dtmanager`	CNCF-Fuzzing repo
`FuzzExtractMessage`	`github.com/kubeedge/kubeedge/cloud/pkg/cloudhub/servers/udsserver`	CNCF-Fuzzing repo
`FuzzextractMessage`	`github.com/kubeedge/kubeedge/cloud/pkg/csidriver`	CNCF-Fuzzing repo
`FuzzParseKey`	`github.com/kubeedge/kubeedge/pkg/metaserver`	CNCF-Fuzzing repo

`FuzzReadMessageFromTunnel`	`github.com/kubeedge/kubeedge/pkg/stream`	CNCF-Fuzzing repo
`FuzzVolumeRegExp`	`github.com/kubeedge/kubeedge/cloud/pkg/cloudhub/handler`	CNCF-Fuzzing repo
`FuzzMqttPublish`	`github.com/kubeedge/kubeedge/edge/pkg/eventbus/mqtt`	CNCF-Fuzzing repo
`FuzzRuleContains`	`github.com/kubeedge/kubeedge/cloud/pkg/router/utils`	CNCF-Fuzzing repo

FuzzLaneReadMessage

Target:
`github.com/kubeedge/viaduct/pkg/lane.(*QuicLane).ReadMessage(*model.Message)`

Description: `FuzzLaneReadMessage` creates a new translator and encodes a pseudo-randomized message. It then instantiates a `*QuickLane{}`, creates a few mocks and finally calls `(*QuicLane).ReadMessage(*model.Message)` with the pseudo-randomized message.

FuzzdealTwinActions

Targets: This fuzzer targets the following APIs all in `github.com/kubeedge/kubeedge/edge/pkg/devicetwin/dtmanager`:

- `dealTwinUpdate()`
- `dealTwinGet()`
- `dealTwinSync()`
- `dealDeviceAttrUpdate()`
- `dealDeviceStateUpdate()`
- `dealSendToCloud()`
- `dealSendToEdge()`
- `dealLifeCycle()`
- `dealConfirm()`
- `dealMembershipGet()`
- `dealMembershipUpdate()`
- `dealMembershipDetail()`

Description: The fuzzer creates a device string, content bytes. It then creates a message with the content bytes. Finally, it selects one of the deal operations and executes it.

FuzzExtractMessage

Target: `github.com/kubeedge/kubeedge/cloud/pkg/cloudhub/servers/udsserver.ExtractMessage()`

Description: Passes a pseudo-random string to the target API.

FuzzextractMessage
Target: `github.com/kubeedge/kubeedge/cloud/pkg/csidriver.extractMessage()`

Description: Passes a pseudo-random string to the target API.

FuzzParseKey
Target: `github.com/kubeedge/kubeedge/pkg/metaserver.ParseKey()`

Description: Passes a pseudo-random string to the target API.

FuzzReadMessageFromTunnel
Target: `github.com/kubeedge/kubeedge/pkg/stream.ReadMessageFromTunnel()`

Description: This fuzzer creates an io.Reader with a pseudo-random buffer and passes it to the target API.

FuzzVolumeRegExp
Target: `github.com/kubeedge/kubeedge/cloud/pkg/cloudhub/handler.VolumeRegExp()`

Description: Tests whether malicious strings can cause disruption when executing the regex.

FuzzMqttPublish
Target: `github.com/256dpi/gomqtt` (3rd party dependency).

Description: This is a fuzzer that was written to test a 3rd party dependency. The fuzzer sets up a server and connects to it by way of a client implemented by gomqtt itself. This is not the same client that KubeEdge uses. It then subscribes to a topic and publishes a payload consisting of pseudo-random bytes provided by the fuzzer.

FuzzRuleContains
Target: `github.com/kubeedge/kubeedge/cloud/pkg/router/utils.RuleContains()`

Description: Passes two pseudo-random strings to the target API.

Threat model formalisation

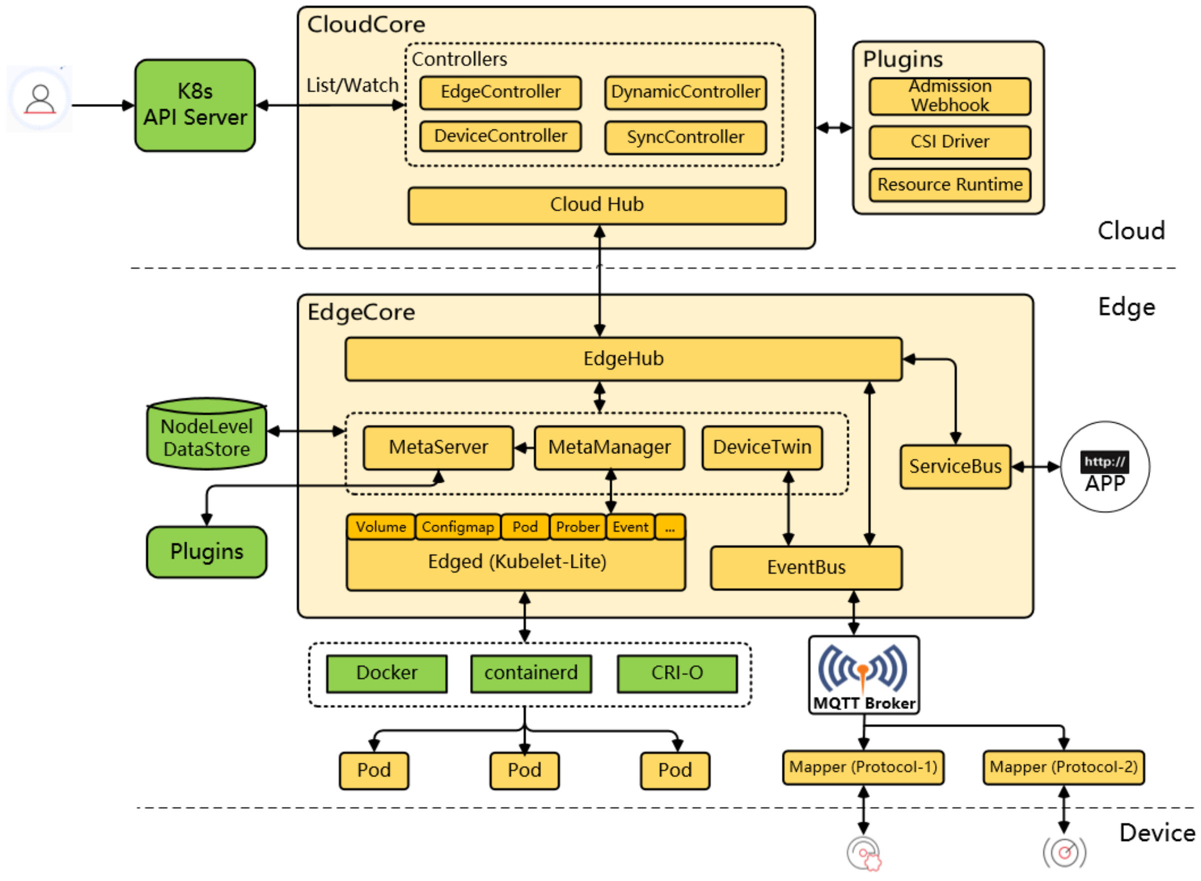
In this section we outline the threat modelling of KubeEdge. The goal of this is to construct an understanding of KubeEdge in order to establish a suitable attack surface that can be used throughout the engagement, both as part of the auditing and the fuzzing integration.

In the context of attack surface enumeration we are interested in understanding the various angles in which a potential adversary can attack the system. In the model we do not focus on a specific threat actor but use a common set of Kubernetes threat actors:

- External malicious attackers, representing actors that access KubeEdge from outside.
- Internal attackers, including inadvertent internal actors who accidentally cause issues.
- Supply chain attackers, representing attackers that subvert components of the KubeEdge software supply chain.

KubeEdge trust architecture

In this section we will outline the trust components of KubeEdge and the boundaries between these trust components. We do this after having reviewed the system from a perspective of understanding its architecture, including code review and documentation review. We construct the trust components by way of the KubeEdge architectural diagram, which is shown in the following figure:



The system is, from a software perspective, composed of two main parts: *Cloud* and *Edge*. The *Device* part in the above diagram does not involve any KubeEdge specific software but rather communicates with the Edge KubeEdge software by way of MQTT.

In order to understand the potential attack scope and the severity of a given attack we separate the system into components of trust. A component of trust specifically means a shared level of trust over data and a shared level of authority in terms of the system functioning properly. We then match these components with the potential actors and use this as a reference on classifying risk throughout the system.

The risk classification will be used to guide severity of potential issues as well as identify where high-severity issues may exist. In this sense, the goal of establishing the trust components is to structurally guide the manual auditing and fuzzing of this security audit towards where security issues may exist.

We divide the system into the following components of trust:

- CloudCore
- EdgeCore
- HTTP server part of EdgeCore
- MQTT broker part of EdgeCore that communicates with Devices
- Edged part EdgeCore which involves running of pods and, thus, containers.

We isolate these parts into components because they form boundaries in the system where distinct trust relationships meet:

- CloudCore is connected to EdgeCore by way of EdgeHub.
- The HTTP server in EdgeCore is exposed locally on the system.
- The MQTT broker part of EdgeCore accepts inputs from the Devices. Privileges flow from low to high in that an attacker in control of a device should not be able to cause adversarial affect on EdgeCore.
- The Edged part of EdgeCore handles the running of Pods.

We will go into more details on these trust relationships in the following section where we outline the separate trust entities at the end of each boundary. We will use this to enumerate the attack surface from a high-level perspective.

KubeEdge attack surface enumeration

In this section we use the trust relationships above to enumerate the possible attack surface.

- **CloudCore connection to EdgeCore**. EdgeCore has a separate level of authority in the overall system, in that an attacker in control of EdgeCore should not be able to negatively affect other edge nodes, e.g. by way of manipulation of CloudCore. In this sense the trust relationship flows from low to high in that EdgeCore has lower authority over KubeEdge than CloudCore, and CloudCore should be considered the highest level of trust in the KubeEdge ecosystem.
- **HTTP server connection to EdgeCore**. The HTTP server in EdgeCore exposes EdgeCore locally on the system and the applications on the local system do not have

a high level of authority over the full KubeEdge cluster. In this context, EdgeCore should be protected against attempts from the local system to thwart KubeEdge. In this sense the trust relationship flows from low to high in that the local applications have lower authority over the KubeEdge ecosystem than EdgeCore.

- **MQTT broker connection to EdgeCore**. The Devices can reach EdgeCore by way of the MQTT broker. There is a trust boundary here where the Devices themselves should not be able to negatively affect the overall KubeEdge system. For example, if an attacker controls a device, the overall system should be protected against possible attacks from this device. In this sense the trust relationship flows from low to high in that the devices have lower authority over the KubeEdge system than EdgeCore.
- **Edged**. Edged handles the running of Pods. The containers in these pods have low authority over the KubeEdge system and privileges thus flow from low to high. An attacker in control of some components in the containers should not be able to grow a foothold of more of the KubeEdge system.

Each of these trust boundaries are a potential entrypoint for a given attack and should be considered when auditing the security posture of KubeEdge. Any behavior that is counter to these trust boundaries should be considered a security risk and should be raised with the KubeEdge team.

EdgeCore: MQTT Broker and ServiceBus security

As a default, EdgeCore trusts all incoming data it receives via ServiceBus and the MQTT Broker. However, because these will often be deployed in untrusted environments and contexts, we include a section that goes further into detail regarding these two parts of EdgeCore.

The MQTT Broker and the ServiceBus receive data from sources that are often not manufactured or maintained by the KubeEdge cluster admin. Furthermore, the cluster admin will not have access to the source code of either the apps that connect to the ServiceBus or the software that runs on the devices that connect to the MQTT broker. Therefore, while EdgeCore trusts input to the ServiceBus and the MQTT Broker, the external services connecting to the ServiceBus and the MQTT Broker expose a critical attack surface that malicious actors will attempt to gain control over as a medium to get control over EdgeCore.

In this section we detail the implications of this attack surface. Specifically, we consider the implications of EdgeCore being exposed to potentially malicious actors, and which impact malicious actors would seek.

The security impact of the ServiceBus and the MQTT Broker can be separated into the following metrics:

Confidentiality
KubeEdge assumes that requests sent to the ServiceBus and the MQTT Broker are authorised and authenticated. If an attacker is able to send requests to the ServieBus and the MQTT Broker, it is assumed that the attacker already has full admin rights over the node.

However, even in the case of full admin rights of an edge node, the attacker should not be able to escalate privileges to admin on the Cloud side.

Integrity
KubeEdge has two positions with regards to the data it receives via ServiceBus and the MQTT Broker:

1. KubeEdge does not make any distinction between correct and incorrect data. Control over an edge node, for example through control over the Devices connecting to EventBus or Apps connecting to ServiceBus, would allow an attacker to manipulate parts of the integrity of the data sent to ServiceBus and EventBus.
2. Control over an edge node should not allow an attacker to mask from which 3rd-party app or device a given request is coming from. An admin on the cloud side must be able to filter incorrect data.

Availability
No incoming requests to the ServiceBus and MQTT Broker, even if these were controlled by a malicious actor, should cause denial of service to EdgeCore or CloudCore.

Implications
Because of this level of trust prescribed to requests coming from the ServiceBus and the MQTT Broker, the following implications can be noted for the security of KubeEdge:

1. **Ensure proper configuration of nodes.** If an attacker has access to a device or an app (app here refers to App in the figure, and is an application on the local system), they are able to manipulate parts of the incoming data to the cluster but should not be able to cause denial of service for either an edge node or the cloud. For this reason, KubeEdge users should be careful to correctly configure the access controls of their edge nodes.
2. **Update 3rd party libraries.** Special attention should be paid to ensuring deployments of the latest security updates on all software on both devices and 3rd-party apps on edge nodes. This is always a general guideline, however, it is important to highlight in this case because local applications on the node or devices provide a way to laterally attack KubeEdge.
3. **Physical attackers may be likely.** There are KubeEdge use cases which may expose some devices in a manner where they are at high risk of physical attacks by malicious actors. This could be the result of onsite control obtained by both external and internal actors. The outcome of physical control of a device is limited to the ability to manipulate the integrity of the data coming from that device. Physical control of a device must not allow an attacker to complete a Denial of Service attack targeted either EdgeCore or CloudCore.

EdgeCore: MetaServer

The MetaServer is a new addition to the KubeEdge. It starts a HTTP server and acts as an edge api-server for Kubernetes operators. It proxies the Kubernetes resource request to the dynamic controller in the cloud, and if the node is offline, the MetaServer will request the edge local storage for a workaround.

The MetaServer is a highly trusted piece of the KubeEdge architecture. This makes it susceptible to attention from attackers that would seek to control the operators communicating with the MetaServer. Obtaining control of the operator would allow attackers to perform several attacks along multiple threat vectors, for example:

- An attacker could send “Create” requests to create new objects to make the victim consume excessive computational resources.
- An attacker could retrieve information about the cluster in an ongoing, stealthy effort to collect intelligence about its victim.
- An attacker could delete resources to deny availability of expected objects.
- An attacker could modify resources to alter integrity of the data in the cluster.

As such, it is of importance that users make sure that the operator(s) communicating with the MetaServer are secure and correctly configured. We also recommend that expectations are set for these operators in terms of their security processes and posture. It is worth noting here that the MetaServer is disabled by default, and any security risks imposed on deployment are only relevant, once it is enabled.

Issues found

In this section we outline the security issues found during the auditing of KubeEdge.

The following CVEs were assigned issues found as part of this audit:

#	Issue	CVE	Severity	Fixed
1	ADA-KE-01: Edge ServiceBus module: DoS by exhausting memory of node with http request containing large body	CVE-2022-31073	Moderate	Y
2	ADA-KE-02: Cloud AdmissionController component: DoS by exhausting memory of node with http request containing large body	CVE-2022-31074	Moderate	Y
3	ADA-KE-03: CloudCore UDS Server: Malicious Message can crash CloudCore	CVE-2022-31076 and CVE-2022-31077	Moderate	Y
4	ADA-KE-04: Deprecated 3rd party libraries		Low	Y
5	ADA-KE-05: DoS when signing the CSR from EdgeCore	CVE-2022-31075	Moderate	Y
6	ADA-KE-06: InsecureSkipVerify: true unsuitable for production		Low	N
7	ADA-KE-7: Use of weak cryptographic primitive		Informational	Y
8	ADA-KE-8: Missing error check for unsafe method		Low	Y
9	ADA-KE-9: CloudCore Router: Large HTTP response can exhaust memory in REST handler	CVE-2022-31078	Moderate	Y
10	ADA-KE-10: Cloud Stream and Edge Stream: DoS from large stream message	CVE-2022-31079	Moderate	Y
11	ADA-KE-11: Websocket Client in package Viaduct: DoS from large response message	CVE-2022-31080	Moderate	Y
12	ADA-KE-12: Possible type confusions		Low	Y

ADA-KE-01: Edge ServiceBus module: DoS by exhausting memory of node with http request containing large body

Severity	Moderate
Difficulty	High
Target	Edge ServiceBus
Fix	https://github.com/kubeedge/kubeedge-ghsa-vwm6-qc77-v2rh/pull/1

The ServiceBus server may be susceptible to a DoS attack if an HTTP request containing a large body is sent to it.

The issue is found in the `buildBasicHandler()` API which builds the handler function for the ServiceBus server. This handler reads the requests body entirely into memory on the marked line below:

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/servicebus/servicebus.go#L235

```
func buildBasicHandler(timeout time.Duration) http.Handler {
      return http.HandlerFunc(func(w http.ResponseWriter, req
*http.Request) {
            sReq := &serverRequest{}
            sResp := &serverResponse{}
            byteData, err := io.ReadAll(req.Body)
```

ADA-KE-02: Cloud AdmissionController component: DoS by exhausting memory of node with http request containing large body

Severity	Moderate
Difficulty	High
Target	Cloud Admissioncontroller
Fix	https://github.com/kubeedge/kubeedge-ghsa-vwm6-qc77-v2rh/pull/1

Several endpoints in the Cloud Admissioncontroller may be susceptible to a DoS attack if an HTTP request containing a large body is sent to it.

The issue is found in the `serve()` API which handles incoming requests for several endpoints in the Admissioncontroller. This API reads the requests body entirely into memory on the marked line below:

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/admissioncontroller/common.go#L66

```
func serve(w http.ResponseWriter, r *http.Request, hook hookFunc) {
	var body []byte
	if r.Body != nil {
		if data, err := io.ReadAll(r.Body); err == nil {
			body = data
		}
	}
```

`serve()` is used the following places:

- https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/admissioncontroller/admit_devicemodel.go#L58
- https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/admissioncontroller/admit_rule.go#L143
- https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/admissioncontroller/admit_ruleendpoint.go#L52
- https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/admissioncontroller/mutate_offlinemigration.go#L67

ADA-KE-03: Nil-pointer dereferences can crash multiple packages in CloudCore

Severity	Moderate
Difficulty	High
Target	Multiple
Fix	https://github.com/kubeedge/kubeedge/pull/3899

An issue was found by the `FuzzextractMessage` fuzzer, whereby a malicious string could crash CloudCore. The issue was reported as a build failure by OSS-Fuzz: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=47779. Upon inspecting the logs of the failed build, it became evident that this was not a build failure, but to OSS-Fuzz it looked like one, because the fuzz harness found the bug within a few iterations.
The issue existed because of passing a double pointer to `json.Unmarshal()`. If `json.Unmarshal()` is passed the bytes `[]byte{"n", "u", "l", "l"}` as its first parameter and a double pointer as its second, the struct to which the buffer should be unmarshalled (passed as the second parameter) will be `nil`.
The issue was found in the UDS Server of the CloudHub and could be used to crash CloudCore.
Two CVEs were assigned this issue:

1. CVE-2022-31076
2. CVE-2022-31077

The double pointer is passed on the line marked below:
https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/cloudhub/servers/udsserver/server.go#L58

```
func ExtractMessage(context string) (*model.Message, error) {
	if context == "" {
		return nil, errors.New("failed with error: context is 
empty")
	}

	var msg *model.Message
	err := json.Unmarshal([]byte(context), &msg)
	if err != nil {
		return nil, err
	}

	return msg, nil
}
```

Upon further examination, it was found that this code pattern was used in several other places in the KubeEdge source code. The fix (https://github.com/kubeedge/kubeedge/pull/3899) corrected 9 uses of double pointers passed to `json.Unmarshal()`.

A similar issue was found in another open source project that Ada Logics previously contributed security work to. See https://adalogics.com/blog/fuzzing-istio-cve-CVE-2022-23635 for more info which describes the root cause in more detail.

ADA-KE-04: Deprecated 3rd party libraries

Severity	Low
Difficulty	High
Target	Multiple
Fix	https://github.com/kubeedge/kubeedge/pull/3972 Issues added upstream regarding transitive dependencies: • https://github.com/container-storage-interface/spec/issues/516 • https://github.com/distribution/distribution/issues/3674

A manual review of KubeEdges 3rd party dependencies was made. 2 factors were considered. First, we checked for use of any deprecated dependencies. Next, we checked for use of any dependencies that were not of the latest release version. Using the latest versions of 3rd party libraries is important, as new releases can include security updates that could affect the security of KubeEdge.

Deprecated libraries
3 projects in KubeEdges dependency tree were found to be deprecated or unmaintained.

- github.com/golang/protobuf
- github.com/gorilla/mux (see https://github.com/gorilla/mux/issues/659)
- github.com/paypal/gatt (see https://github.com/paypal/gatt/issues/75). gatt is only used for testing.

Non-latest dependencies

Name	Used version	Latest version
`github.com/256dpi/gomqtt`	v0.10.4	v0.14.4
`github.com/blang/semver`	v3.5.1	v4.0.0
`github.com/container-storage-interface/spec`	v1.5.0	v1.6.0
`github.com/eclipse/paho.mqtt.golang`	V1.2.0	v1.3.5
`ithub.com/golang-jwt/jwt`	v3.2.2	v4.4.1
`github.com/google/uuid`	v1.2.0	v1.3.0
`github.com/gorilla/websocket`	v1.4.2	v1.5.0
`github.com/kubernetes-csi/csi-lib-utils`	v.0.6.1	v0.11.0
`github.com/prometheus/client_golang`	v1.11.0	1.12.2
`github.com/spf13/cobra`	v1.2.1	v1.4.0

`golang.org/x/net`	v0.0.0-20211209124913-491a49abca63	v0.0.0-20220526153639-5463443f8c37
`google.golang.org/grpc`	v1.42.0	v1.46.2
`helm.sh/helm/v3`	v3.7.2	v3.9.0
`k8s.io/api`	v0.22.6	v0.24.1
`k8s.io/apiextensions-apiserver`	v0.22.6	v0.24.1
`k8s.io/apimachinery`	v0.22.6	v0.24.1
`k8s.io/apiserver`	v0.22.6	v0.24.1
`k8s.io/cli-runtime`	v0.22.6	v0.24.1
`k8s.io/client-go`	v0.22.6	v0.24.1
`k8s.io/cloud-provider`	v0.22.6	v0.24.1
`k8s.io/cluster-bootstrap`	v0.22.6	v0.24.1
`k8s.io/code-generator`	v0.22.6	v0.24.1
`k8s.io/component-base`	v0.22.6	v0.24.1
`k8s.io/cri-api`	v0.22.6	v0.24.1
`k8s.io/csi-translation-lib`	v0.22.6	v0.24.1
`k8s.io/klog/v2`	v2.9.0	v2.60.1
`k8s.io/kube-openapi`	v0.0.0-20211115234752-e816edb12b65	v0.0.0-20220413171646-5e7f5fdc6da6
`k8s.io/kube-scheduler`	v0.22.6	v0.24.1
`k8s.io/kubelet`	v0.22.6	v0.24.1
`k8s.io/kubernetes`	v1.22.6	v1.24.1
`k8s.io/mount-utils`	v0.22.6	v0.24.1
`k8s.io/utils`	v0.0.0-20210819203725-bdf08cb9a70a	v0.0.0-20220210201930-3a6ce19ff2f9
`sigs.k8s.io/apiserver-network-proxy`	v0.0.27	v0.0.9-klog-v1
`sigs.k8s.io/apiserver-network-proxy/konnectivity-client`	v0.0.27	v0.0.9-klog-v1

`sigs.k8s.io/yaml`	v1.2.0	v1.3.0

We recommend never to use deprecated libraries. Furthermore, unless impossible, 3rd-party dependencies should be updated to the latest to include upstream reliability and security fixes.

ADA-KE-05: DoS when signing the CSR from EdgeCore

Severity	Moderate
Difficulty	High
Target	https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/cloudhub/servers/httpserver/server.go#L172
Fix	https://github.com/kubeedge/kubeedge-ghsa-vwm6-qc77-v2rh/pull/1

EdgeCore may be susceptible to a DoS attack on CloudHub if an attacker was to send a well-crafted HTTP request to `/edge.crt`.

The root cause of this issue is that `signEdgeCert()` reads the entire HTTP request into memory.

The endpoint `/edge.crt` is initiated here:

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/cloudhub/servers/httpserver/server.go#L44

```
func StartHTTPServer() {
	router := mux.NewRouter()
	router.HandleFunc(constants.DefaultCertURL,
edgeCoreClientCert).Methods(http.MethodGet)
	router.HandleFunc(constants.DefaultCAURL, getCA).Methods(http.MethodGet)
```

`edgeCoreClientCert()` is the handler for the endpoint `/edge.crt` and is implemented here:

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/cloudhub/servers/httpserver/server.go#L84

```
func edgeCoreClientCert(w http.ResponseWriter, r *http.Request) {
	if cert := r.TLS.PeerCertificates; len(cert) > 0 {
		if err := verifyCert(cert[0]); err != nil {
			klog.Errorf("failed to sign the certificate for edgenode:
%s, failed to verify the certificate", r.Header.Get(constants.NodeName))
			w.WriteHeader(http.StatusUnauthorized)
			if _, err := w.Write([]byte(err.Error())); err != nil {
				klog.Errorf("failed to write response, err: %v", err)
			}
		} else {
			signEdgeCert(w, r)
		}
		return
	}
	if verifyAuthorization(w, r) {
```

```
            signEdgeCert(w, r)
      } else {
            klog.Errorf("failed to sign the certificate for edgenode: %s,
invalid token", r.Header.Get(constants.NodeName))
      }
}
```

If an attacker can send a well-crafted HTTP request that proceeds into `signEdgeCert()` above, and that request has a very large body, that request could crash the http service through a memory exhaustion vulnerability. The requests body is being read into memory on the line below, and a body that it larger than the available memory could lead to a successful attack:

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/cloudhub/servers/httpserver/server.go#L172

```
func signEdgeCert(w http.ResponseWriter, r *http.Request) {
      csrContent, err := io.ReadAll(r.Body)
      if err != nil {
            klog.Errorf("fail to read file when signing the cert for
edgenode:%s! error:%v", r.Header.Get(constants.NodeName), err)
            return
      }
      csr, er
...
```

Because the request would have to make it through `verifyAuthorization()`, only authorized users could perform this attack.

ADA-KE-06: InsecureSkipVerify: true unsuitable for production

Severity	Low
Difficulty	High
Target	Multiple
Fix	Not fixed. An issue has been created on Github to track that this gets fixed in future versions of KubeEdge: https://github.com/kubeedge/kubeedge/issues/4001

The `InsecureSkipVerify: true` setting is unsuited for production as per the Golang documentation for the TLS package:

"InsecureSkipVerify controls whether a client verifies the server's certificate chain and host name. If InsecureSkipVerify is true, crypto/tls accepts any certificate presented by the server and any host name in that certificate. In this mode, TLS is susceptible to machine-in-the-middle attacks unless custom verification is used. This should be used only for testing or in combination with VerifyConnection or VerifyPeerCertificate."
https://pkg.go.dev/crypto/tls

`InsecureSkipVery: true` is used in the following places:

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/eventbus/common/util/common.go#L86

```
	} else {
		tlsConfig = &tls.Config{InsecureSkipVerify: true,
ClientAuth: tls.NoClientCert}
	}
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edgestream/edgestream.go#L87

```
	tlsConfig := &tls.Config{
		InsecureSkipVerify: true,
		Certificates:       []tls.Certificate{cert},
	}
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edgehub/common/http/http.go#L38

```
	transport := &http.Transport{
		DialContext: (&net.Dialer{
			Timeout:   connectTimeout,
                KeepAlive: keepaliveTimeout,
            }).DialContext,
            MaxIdleConnsPerHost:   maxIdleConnectionsPerHost,
            ResponseHeaderTimeout: responseReadTimeout,
            TLSClientConfig:       &tls.Config{InsecureSkipVerify: 
true},
        }
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edgehub/common/http/http.go#L60

```
        tr := &http.Transport{
            TLSClientConfig: &tls.Config{
                RootCAs:      pool,
                Certificates: []tls.Certificate{cliCrt},
                MinVersion:   tls.VersionTLS12,
                CipherSuites: []uint16{
                    tls.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
                },
                InsecureSkipVerify: true}, /*Now we need set it true*/
        }
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edgehub/clients/quicclient/quicclient.go#L62

```
        tlsConfig := &tls.Config{
            RootCAs:            pool,
            Certificates:       []tls.Certificate{cert},
            InsecureSkipVerify: true,
        }
```

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/router/utils/http/http.go#L38

```
        transport := &http.Transport{
            DialContext: (&net.Dialer{
                Timeout:   connectTimeout,
                KeepAlive: keepaliveTimeout,
            }).DialContext,
            MaxIdleConnsPerHost:   maxIdleConnectionsPerHost,
            ResponseHeaderTimeout: responseReadTimeout,
            TLSClientConfig:       &tls.Config{InsecureSkipVerify: 
true},
        }
```

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/router/utils/http/http.go#L45

```
	tr := &http.Transport{
		TLSClientConfig: &tls.Config{
			RootCAs:      pool,
			Certificates: []tls.Certificate{cliCrt},
			MinVersion:   tls.VersionTLS12,
			CipherSuites: []uint16{
				tls.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
			},
			InsecureSkipVerify: true}, /*Now we need set it true*/
	}
```

ADA-KE-7: Use of weak cryptographic primitive

Severity	Informational
Difficulty	n/a
Target	CloudCore Dynamic Controller
Fix	WontFix

As is noted by the Golang documentation:

"MD5 is cryptographically broken and should not be used for secure applications."
https://pkg.go.dev/crypto/md5

MD5 encryption was used in CloudCores Dynamic Controller to generate an Application ID. Because of that, the issue is scored as "Informational".

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/dynamiccontroller/application/application.go#L145

```
func (a *Application) Identifier() string {
	if a.ID != "" {
		return a.ID
	}
	b := []byte(a.Nodename)
	b = append(b, []byte(a.Key)...)
	b = append(b, []byte(a.Verb)...)
	b = append(b, a.Option...)
	b = append(b, a.ReqBody...)
	a.ID = fmt.Sprintf("%x", md5.Sum(b))
	return a.ID
}
```

ADA-KE-8: Missing error check for unsafe method

Severity	Low
Difficulty	High
Target	Multiple
Fix	https://github.com/kubeedge/kubeedge/pull/3975

File close operations can return an error that is not checked in deferred calls in KubeEdge. This can lead to unexpected behaviour and potentially allow attackers to create files on the machine and at the same time avoid an audit trail.

https://github.com/kubeedge/kubeedge/blob/master/keadm/cmd/keadm/app/cmd/util/common.go#L572

```
func computeSHA512Checksum(filepath string) (string, error) {
	f, err := os.Open(filepath)
	if err != nil {
		return "", err
	}
	defer f.Close()
```

https://github.com/kubeedge/kubeedge/blob/master/keadm/cmd/keadm/app/cmd/util/common.go#L724

```
			// add file to tar
			srcFile, err := os.Open(file)
			if err != nil {
				return err
			}
			defer srcFile.Close()
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edged/volume/csi/csi_util.go#L63

```
	file, err := os.Create(dataFilePath)
	if err != nil {
		klog.Error(log("failed to save volume data file %s: %v",
dataFilePath, err))
		return err
	}
	defer file.Close()
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edged/volume/csi/csi_util.go#L83

```
	file, err := os.Open(dataFileName)
	if err != nil {
		klog.Error(log("failed to open volume data file [%s]: %v",
dataFileName, err))
		return nil, err
	}
	defer file.Close()
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/common/util/config.go#L94

```
	file, err := os.Create(path)
	if err != nil {
		return err
	}

	defer file.Close()
```

ADA-KE-9: CloudCore Router: Large HTTP response can exhaust memory in REST handler

Severity	Moderate
Difficulty	High
Target	`github.com/kubeedge/kubeedge/blob/master/cloud/pkg/router/listener.(rh *RestHandler).httpHandler(w http.ResponseWriter, r *http.Request)`
Fix	https://github.com/kubeedge/kubeedge-ghsa-vwm6-qc77-v2rh/pull/1

KubeEdge does not impose a limit on the size of responses to requests made by the REST handler. An attacker could use this vulnerability to make a request that will return an http response with a large body and cause DoS of CloudCore.

In the `httpHandler()` API, the rest handler makes a request to a pre-specified handle. The handle will return an http response that is then read entirely into memory.

The line which can be exploited to invoke a DoS is marked below:
https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/router/listener/http.go

```
func (rh *RestHandler) httpHandler(w http.ResponseWriter, r
*http.Request) {
	...
	if isNodeName(uriSections[1]) {
		params := make(map[string]interface{})
		msgID := uuid.New().String()
		params["messageID"] = msgID
		params["request"] = r
		params["timeout"] = rh.restTimeout
		params["data"] = b

		v, err := handle(params)
		if err != nil {
			klog.Errorf("handle request error, msg id: %s, err:
%v", msgID, err)
			return
		}
		response, ok := v.(*http.Response)
		if !ok {
			klog.Errorf("response convert error, msg id: %s",
msgID)
			return
		}
		body, err := io.ReadAll(response.Body)
```

```
            if err != nil {
                  klog.Errorf("response body read error, msg id: %s,
reason: %v", msgID, err)
                  return
            }
      ...
}
```

This attack would require a handle to first be created for the `RequestURI` passed in the http request to execute this attack. Naturally, this makes it highly difficult to complete. It would require a high level of privileges to create handles and could not be executed on its own.

ADA-KE-10: Cloud Stream and Edge Stream: DoS from large stream message

Severity	Moderate
Difficulty	High
Target	TunnelServer
Fix	https://github.com/kubeedge/kubeedge-ghsa-vwm6-qc77-v2rh/pull/1

A DoS issue exists in the `ReadMessageFromTunnel()` api. The API reads the entire message into memory without imposing a limit on the size of this message. An attacker can exploit this by sending a large message to exhaust memory and cause a DoS.

```go
func ReadMessageFromTunnel(r io.Reader) (*Message, error) {
	buf := bufio.NewReader(r)
	connectID, err := binary.ReadUvarint(buf)
	if err != nil {
		return nil, err
	}
	messageType, err := binary.ReadUvarint(buf)
	if err != nil {
		return nil, err
	}
	data, err := io.ReadAll(buf)
	if err != nil {
		return nil, err
	}
	klog.V(6).Infof("Receive Tunnel message connectID %d messageType %s 
data:%v string:[%v]",
		connectID, MessageType(messageType), data, string(data))
	return &Message{
		ConnectID:   connectID,
		MessageType: MessageType(messageType),
		Data:        data,
	}, nil
}
```

ADA-KE-11: Websocket Client in package Viaduct: DoS from large response message

Severity	Moderate
Difficulty	High
Target	https://github.com/kubeedge/kubeedge/blob/master/staging/src/github.com/kubeedge/viaduct/pkg/client/ws.go#L69
Fix	https://github.com/kubeedge/kubeedge-ghsa-vwm6-qc77-v2rh/pull/1

A large response received by the viaduct `WSClient` can cause a DoS from memory exhaustion. The entire body of the response is being read into memory which could allow an attacker to send a request that returns a response with a large body.

```
func (c *WSClient) Connect() (conn.Connection, error) {
	header := c.exOpts.Header
	header.Add("ConnectionUse", string(c.options.ConnUse))
	wsConn, resp, err := c.dialer.Dial(c.options.Addr, header)
	if err == nil {
		klog.Infof("dial %s successfully", c.options.Addr)

		// do user's processing on connection or response
		if c.exOpts.Callback != nil {
			c.exOpts.Callback(wsConn, resp)
		}
		return conn.NewConnection(&conn.ConnectionOptions{
			ConnType: api.ProtocolTypeWS,
			ConnUse:  c.options.ConnUse,
			Base:     wsConn,
			Consumer: c.options.Consumer,
			Handler:  c.options.Handler,
			CtrlLane: lane.NewLane(api.ProtocolTypeWS, wsConn),
			State: &conn.ConnectionState{
				State:   api.StatConnected,
				Headers: c.exOpts.Header.Clone(),
			},
			AutoRoute: c.options.AutoRoute,
		}), nil
	}

	// something wrong!!
	var respMsg string
	if resp != nil {
		body, errRead := io.ReadAll(resp.Body)
		if errRead == nil {
```

```
                    respMsg = fmt.Sprintf("response code: %d, response body:
%s", resp.StatusCode, string(body))
             } else {
                    respMsg = fmt.Sprintf("response code: %d", resp.StatusCode)
             }
             resp.Body.Close()
      }
      klog.Errorf("dial websocket error(%+v), response message: %s", err,
respMsg)

      return nil, err
}
```

ADA-KE-12: Possible type confusions

Severity	Low
Difficulty	High
Target	Multiple
Fix	https://github.com/kubeedge/kubeedge/pull/3976

Several possible type confusions were discovered in the KubeEdge code base. These should be avoided by checking for an error when type casting.

The following type confusions were found:
https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/eventbus/eventbus.go#L130-L131

```
case messagepkg.OperationMessage:
	body, ok := accessInfo.GetContent().(map[string]interface{})
	if !ok {
		klog.Errorf("Message is not map type")
		continue
	}
	message := body["message"].(map[string]interface{})
	topic := message["topic"].(string)
```

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/router/provider/eventbus/eventbus.go#L98

```
func (*EventBus) Forward(target provider.Target, data interface{})
(response interface{}, err error) {
	message := data.(*model.Message)
```

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/router/provider/servicebus/servicebus.go#L83

```
func (sb *ServiceBus) Forward(target provider.Target, data interface{})
(response interface{}, err error) {
	message := data.(*model.Message)
```

https://github.com/kubeedge/kubeedge/blob/master/cloud/pkg/router/listener/http.go#L81

```
func (rh *RestHandler) matchedPath(uri string) (string, bool) {
	var candidateRes string
	rh.handlers.Range(func(key, value interface{}) bool {
		pathReg := key.(string)
```

https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/devicetwin/dtmanager/communicate.go#L77

```
func dealSendToEdge(context *dtcontext.DTContext, resource string, msg
interface{}) error {
      beehiveContext.Send(dtcommon.EventHubModule,
*msg.(*model.Message))
      return nil
}
```

Code complexity

In this section we discuss some observations made during the audit about the code complexity of KubeEdge. The information presented in this section does not present any immediate security issues, and it should be considered informational.

Code complexity can affect the security posture of a software product in different ways. Complex code makes maintenance more difficult, and it can add an overhead when patching security issues. It increases the difficulty when looking for and triaging security issues. In this audit, the KubeEdge source code was audited for ways to reduce complexity. Overall, we found that the KubeEdge project is very well written, well documented and easy to approach. We did find a few areas where KubeEdge could be simplified to avoid security issues that arise because of unnecessary complexity.

Pointer arguments

Ada Logics made an assessment of KubeEdges handling of pointer arguments. Two CVEs were found in this audit from nil-pointer dereferences, and because of that we elaborate on KubeEdges general approach to handling pointers passed as function arguments. This section is intended to give the KubeEdge maintainers some ideas to further improve the robustness and reliability of the project.

In our assessment we looked at all the places a pointer to a `Message` is passed as a function argument. The goal of this exercise was to look for possible nil-pointer dereferences in the code base. This was done manually, and findings do not directly impose a security risk. Several cases of possible nil-pointers were found which we present below.

Possible nil-pointers

This table presents functions in which a `nil` value could be passed which could trigger a nil-pointer dereference panic. These do not necessarily impose a risk in terms of security. Even if these do not have security implications now, this might change every time changes are made to the KubeEdge source tree.
These were fixed in https://github.com/kubeedge/kubeedge/pull/4000

File	**Function**
`cloud/pkg/router/listener/message.go`	`(mh *MessageHandler).HandleMessage(message *model.Message)`
`cloud/pkg/router/listener/message.go`	`(mh *MessageHandler).callback(message *model.Message)`
`cloud/pkg/cloudhub/common/model/types.go`	`IsFromEdge(msg *model.Message)`
`cloud/pkg/cloudhub/common/model/types.go`	`IsToEdge(msg *model.Message)`

staging/src/github.com/kubeedge/viaduct/pkg/mux/pattern.go	(pattern *MessagePattern) matchOp(message *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/mux/pattern.go	(pattern *MessagePattern) Match(message *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/conn/quic.go	(conn *QuicConnection) headerMessage(msg *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/conn/quic.go	(conn *QuicConnection) processControlMessage(msg *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/conn/quic.go	(conn *QuicConnection) WriteMessageAsync(msg *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/conn/ws.go	(conn *WSConnection) WriteMessageAsync(msg *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/conn/ws.go	(conn *WSConnection) WriteMessageSync(msg *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/conn/comm.go	(r *responseWriter) WriteResponse(msg *model.Message, content interface{})
staging/src/github.com/kubeedge/viaduct/pkg/translator/message.go	(t *MessageTranslator) protoToModel(src *message.Message, dst *model.Message)
staging/src/github.com/kubeedge/viaduct/pkg/translator/message.go	(t *MessageTranslator) modelToProto(src *model.Message, dst *message.Message)
staging/src/github.com/kubeedge/viaduct/pkg/keeper/synckeeper.go	(k *SyncKeeper) WaitResponse(msg *model.Message, deadline time.Time)
staging/src/github.com/kubeedge/beehive/pkg/core/socket/modules/socket/server.go	(m *Server) processModuleMessage(conn wrapper.Conn, message *model.Message)
edge/pkg/metamanager/client/metaclient.go	(s *send) SendSync(message *model.Message)
edge/pkg/metamanager/process.go	msgDebugInfo(message *model.Message)
edge/pkg/edgehub/common/msghandler/handler.go	(*defaultHandler) Filter(message *model.Message)
edge/pkg/edgehub/common/msghandler/handler.go	(*defaultHandler) Process(message *model.Message, clientHub clients.Adapter)
edge/pkg/devicetwin/dtcontext/dtcontext.go	(dtc *DTContext) BuildModelMessage(group string, parentID string, resource string, operation string, content interface{})

Recommendations

Nil-pointers can be exploitable by an attacker if triggerable by an attacker under the right conditions. Naturally, whether a possible nil-pointer dereference is triggerable by an attacker depends largely on how the function is used. In many cases it is never possible to pass `nil`. As a rule of thumb, we recommend checking pointers for nil at the beginning of the function body. In our analysis of pointer-handling in KubeEdge, we used the Message type as an example to demonstrate an area for improvement.

Unused parameters

Code complexity can be reduced by ensuring that functions only accept parameters that are actually used by the function to which they are passed.

These were fixed in https://github.com/kubeedge/kubeedge/pull/3994

Function name	**URL**	**Unused parameters**
`(c *csiAttacher).waitForVolumeAttachment()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edged/volume/csi/csi_attacher.go#L87	`timeout`
`(h *RegistrationHandler).validateVersions`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edged/volume/csi/csi_plugin.go#L169	`endpoint`
`(e *edged) makeBlockVolumes()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edged/edged_pods.go#L887	`pod`
`detailRequest()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/devicetwin/dtmanager/communicate.go#L129	`msg`
`(cw CommWorker).checkConfirm()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/devicetwin/dtmanager/communicate.go#L150	`msg`
`(p *Proxy).runAdminServer()`	https://github.com/kubeedge/kubeedge/blob/master/edgesite/cmd/edgesite-server/app/server.go#L329	`server`
`runEdgeCore()`	https://github.com/kubeedge/kubeedge/blob/master/keadm/cmd/keadm/app/cmd/util/common.go#L454	`version`

Functions always returning same value

These were fixed in https://github.com/kubeedge/kubeedge/pull/3994

Function name	URL	Return values
`detailRequest()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/devicetwin/dtmanager/communicate.go#L129	First return value (`interface{}`) is always nil
`(cw CommWorker).checkConfirm()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/devicetwin/dtmanager/communicate.go#L150	First return value (`interface{}`) is always nil
`dealTwinDelete()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/devicetwin/dtmanager/twin.go#L370	The return value (`error`) is always nil
`dealMembershipGetInner()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/devicetwin/dtmanager/membership.go#L310	The return value (`error`) is always nil
`(e *edged).setCPUInfo()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edged/edged_status.go#L359	The return value (`error`) is always nil
`(p *Proxy).runAdminServer()`	https://github.com/kubeedge/kubeedge/blob/master/edgesite/cmd/edgesite-server/app/server.go#L329	The return value (`error`) is always nil
`(p *Proxy).runHealthServer()`	https://github.com/kubeedge/kubeedge/blob/master/edgesite/cmd/edgesite-server/app/server.go#L356	The return value (`error`) is always nil
`setConfigDefaults()`	https://github.com/kubeedge/kubeedge/blob/master/pkg/client/clientset/versioned/typed/devices/v1alpha2/devices_client.go#L74	The return value (`error`) is always nil
`setConfigDefaults()`	https://github.com/kubeedge/kubeedge/blob/master/pkg/client/clientset/versioned/typed/reliablesyncs/v1alpha1/reliablesyncs_client.go#L74	The return value (`error`) is always nil
`setConfigDefaults()`	https://github.com/kubeedge/kubeedge/blob/master/pkg/client/clientset/ver	The return value (`error`) is always nil

	sioned/typed/rules/v1/rules_client.go#L74	
`initializeCSINode()`	https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/edged/volume/csi/csi_plugin.go#L257	The return value (`error`) is always nil

Other unnecessary complexity

These were fixed in https://github.com/kubeedge/kubeedge/pull/3994
The Quic and WS connections of the Viaduct package contain logic that could be removed altogether. In both cases, the switch statement can be removed. A comment should be left to explain why the functions look like they do. It could be considered whether `processControlMessage` could be removed entirely.

https://github.com/kubeedge/kubeedge/blob/master/staging/src/github.com/kubeedge/viaduct/pkg/conn/quic.go#L75

```
func (conn *QuicConnection) processControlMessage(msg *model.Message)
error {
	switch msg.GetOperation() {
	case comm.ControlTypeConfig:
	case comm.ControlTypePing:
	case comm.ControlTypePong:
	}
	return nil
}
```

https://github.com/kubeedge/kubeedge/blob/master/staging/src/github.com/kubeedge/viaduct/pkg/conn/ws.go#L60

```
func (conn *WSConnection) processControlMessage(msg *model.Message)
error {
	switch msg.GetOperation() {
	case comm.ControlTypeConfig:
	case comm.ControlTypePing:
	case comm.ControlTypePong:
	}
	return nil
}
```

Appendix

KubeEdge SLSA review

In this section we perform a review of SLSA compliance in KubeEdge. The Supply Chain Levels for Software Artifacts - or SLSA (pronounced “salsa”) - is a security framework for ensuring integrity of software artifacts in the software supply chain. It was introduced in June 2021[1] and is developed to help protect against common supply chain attacks. Recent years saw a number of supply chain attacks targeted the open source environment. Some of these attacks include high-profile software packages that are used by billions of users and in mission-critical deployments. Examples of these attacks are enumerated here: https://slsa.dev/spec/v0.1/threats. Furthermore, cyber attacks targeting open source sotware projects are increasing in volume with some researching indicating a growth of 430% for cyber attacks against open source software in 2020 alone[2].

As a widely deployed open source software project, KubeEdge is exposed to attackers targeting the open source community of software development. Because of that, a SLSA review was carried out as part of this security audit. At the time of this review, few SLSA audits have been carried out. This is due to the recent launch of SLSA. As such, KubeEdge is one of the first projects to carry out a SLSA audit.

The SLSA framework is still in alpha, and the framework is likely to undergo some changes in the near future before beta and subsequently full release.

Executive summary

KubeEdges is developed through GitHub, and releases are built in an automatic build service. This sets the project up for SLSA compliance with a great foundation, and it is recommended that pursuance of compliance is made on top of this foundation.

KubeEdge is doing well in the areas of Source and Build. With a few missing details, the project is close to full compliance.

The major missing part is the provenance. The purpose of the provenance is to verify the production of software artifacts. The provenance provides a verifiable piece of information that assures consumers and users of supply-chain integrity. During KubeEdges SLSA review, no provenance was found to be available. Should KubeEdge pursue full compliance, it is recommended that provenance becomes the primary focus, as this is where most work is required.

[1] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html

[2] https://www.sonatype.com/hubfs/Corporate/Software%20Supply%20Chain/2020/SON_SSSC-Report-2020_final_aug11.pdf#page=7

SLSA Analysis

In this section we iterate over the requirements of SLSA as specified in http://slsa.dev/spec/v0.1/requirements The section begins with a description of justifications on the analysis and ends with an overview of the assessment table with indications of whether KubeEdge meets the requirement or not.

Source

KubeEdge has achieved full compliance in source. The project is maintained in a version controlled environment, Github, with a history that is retained indefinitely. Pull requests not authored by trusted maintainers are approved by two trusted maintainers, whereas pull requests made by trusted maintainers are approved by a single trusted maintainer. KubeEdges contribution guide does not state that commits need to be signed. This could be an area of change for higher integrity.

Build

The KubeEdge build system runs on every pull request. The build system runs in Github actions with a high level of automation. Release artifacts are produced with complete automation, and in ephemeral environments that are procured specifically for the given build and are subsequently discarded.
The build is fully automatic and runs in an environment that is free from interference from the outside world. The environment in which the build runs is provisioned solely for the given build and is not reused. Artifacts are fetched in a trusted control plane, and TLS ensures transport integrity. The build service prevents network access while running the build steps.

Provenance

No provenance was found.

SLSA assessment table

Requirement	L1	L2	L3	L4
Source				
Version controlled	Y	Y	Y	Y
Verified history			Y	Y
Retained indefinitely			Y	Y
Two-person reviewed				Y
Build				
Scripted build	Y	Y	Y	Y
Build Service		Y	Y	Y
Build as code			Y	Y
Ephemeral environment			Y	Y
Isolated			Y	Y

Parameterless				Y
Hermetic				Y
Provenance				
Available	N	N	N	N
Authenticated		N	N	N
Service generated		N	N	N
Non-falsifiable			N	N
Depencies completion				N
Common				
Security				Y
Access				Y
Superusers				Y

Recommendations

This section presents recommendations for KubeEdge to improve its SLSA compliance. Future work will build upon a strong foundation. As of this writing there is scarce documentation on how to achieve SLSA compliance in practice. We recommend the following CloudNativeCon presentation “Securing Your Container Native Supply Chain with SLSA, Github and Tekton” for which there is a recording available here: https://www.youtube.com/watch?v=iZpFtalj4xE

In short, our recommendation are:

- Integrate with Tekton (https://tekton.dev) and Tekton Chains (https://github.com/tektoncd/chains) to achieve SLSA level 2 compliance.
- Integrate the build as part of Github Actions and workflows together with a trusted builder to achieve SLSA compliance level 3.

In the following we provide further recommendations.

Documentation

We recommend that KubeEdge enforces that commits are signed. This includes adding this requirement to the documentation for contributors as well as a test in its CI that tests it. Furthermore, documentation should include guidance on retrieving provenance as well as which steps are taken to comply with SLSA requirements.

Provenance

KubeEdges main focus should be on generating SLSA-compliant provenance and making it available to users. To make this non-falsifiable, which would bring KubeEdge near level 3 compliance, the provenance should be signed. Sigstore[3] can be used to sign and verify provenance data and fulfil this requirement.

[3] https://www.sigstore.dev/

Build
Ensure that releases are built in environments without network access. Once this is satisfied, KubeEdge satisfies the requirements of release builds.

Make compliance a community effort
We recommend making SLSA compliance a community effort. Full compliance requires satisfying many areas of the development and release pipelines, and detailed may be missed. We recommend including a section in KubeEdges documentation, where the assumed compliance is available, and where contributors are invited to verify this and report any findings that may be contrary to the assumed compliance. The assumed compliance can be presented in the form of the SLSA assessment table above.