PRESENTS

Knative security audit

In collaboration with the Knative maintainers, Open Source Technology Improvement Fund and The Linux Foundation

Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>

Date: 4th December 2023

Table of contents

Project Summary

The auditors of Ada Logics were:

Name	Title	Email
Adam Korczynski	Security Engineer, Ada Logics	Adam@adalogics.com
David Korczynski	Security Researcher, Ada Logics	David@adalogics.com

The Knative community members involved in audit were:

Name	Title	Email
Evan Anderson	Knative Maintainer	Evan.k.anderson@gmail.com
David Hadas	Knative Maintainer	Davidh@il.ibm.com

The following facilitators of OSTIF were engaged in the audit:

Name	Title	Email
Derek Zimmer	Executive Director, OSTIF	Derek@ostif.org
Amir Montazery	Managing Director, OSTIF	Amir@ostif.org
Helen Woeste	Project Manager, OSTIF	Helen@ostif.org

Audit Scope

The following assets were in scope of the audit.

Repository	https://github.com/knative/eventing
Language	Go

Repository	https://github.com/knative/serving
Language	Go

Repository	https://github.com/knative/pkg
Language	Go

Repository	https://github.com/knative/func
Language	Go

Repository	https://github.com/knative-extensions/eventing-autoscaler-keda
Language	Go

Repository	https://github.com/knative-extensions/eventing-ceph
Language	Go

Repository	https://github.com/knative-extensions/eventing-couchdb
Language	Go

Repository	https://github.com/knative-extensions/eventing-github
Language	Go

Repository	https://github.com/knative-extensions/eventing-gitlab
Language	Go

Repository	https://github.com/knative-extensions/eventing-istio
Language	Go

Repository	https://github.com/knative-extensions/eventing-kafka-broker
Language	Go

Repository	https://github.com/knative-extensions/eventing-redis
Language	Go

Executive summary

In the fall of 2023, Ada Logics conducted a security audit of Knative in a collaboration between Ada Logics, the Knative maintainers, The Open Source Technology Improvement Fund (OSTIF) and the Cloud Native Computing Foundation (CNCF). The engagement was a holistic security audit with the following goals:

1. Formalize a threat model of the Knative ecosystem.
2. Manually audit the Knative code base for security vulnerabilities of any severity.
3. Assess Knatives supply-chain risk against the SLSA framework.

The main scope of the audit was the Eventing, Serving and Pkg sub-projects with an additional minor focus on the Knative Extensions projects.

The audit found 16 issues ranging from Informational to High severity. Ada Logics reported the found issues ad hoc to the Knative team, who would coordinate that the Knative community fix the issues. The Ada Logics team also helped fix the issues found with patches submitted to the Knative repositories.
The most exciting security finding was a vulnerability in Knative serving, which could allow an attacker with escalated privileges in one Knative pod to cause a denial of service of the compromised Knative deployment. The finding has been assigned CVE-2023-48713 and fixed in Knative Serving v1.12.0 and v1.11.3.

Knative implements a provenance generator that adds in-toto provenance attestations to releases. Knative was maintaining and using this prior to this security, and consumers have been able to verify their Knative artifacts prior to consuming.

At the time of the audit, Knative Serving was not adding the provenance to releases; The maintainers fixed this before the audit completed.

Strategic Recommendations

In this section, we enumerate our strategic recommendations for Knative. We recommend that the Knative community works on these improvements in the long term to improve its security posture over time. They are practical and approachable by maintainers and contributors.

Review Knative's third-party dependencies

Ada Logics found several code issues in third-party dependencies during the audit. Some of these were found in user-exposed APIs. In addition to code errors, we found that several of Knative's third-party dependencies are not actively maintained, making it hard for community contributors to submit patches to fix found issues. For example, the Knative Eventing-Github uses the webhook implementation from the https://github.com/go-playground/webhooks library to receive events from GitHub. From our assessment, https://github.com/go-playground/webhooks does not meet the security standards that Knative requires. We recommend that Knative performs an ongoing review of dependencies to ensure that they 1) are required and 2) that they meet industry best practices. On the first point, whether Knative's dependencies are required, Knative might be importing a whole package to use a small part of the logic, and we recommend assessing whether Knative can implement the same logic without importing a given package. On the second point, Knative can use the Scorecard (https://github.com/ossf/scorecard) project to evaluate the security risk of its dependencies and require third-party dependencies to maintain a high Scorecard score.

Improve SAST tooling for the entire Knative ecosystem

Knative has integrated SAST tools in its core packages, Eventing and Serving. During the audit, Ada Logics ran the same tools against Knative Extensions projects, Func and Security Guard, which revealed true-positive findings. We recommend maintaining the same SAST suite for Knative Extensions projects and Knative Func as Knative Eventing and Serving maintains.

Threat model

In this section, we present the findings of the threat modelling goal of the security audit. We first cover the data flow of the Knative ecosystem, then common threats that Knative and its users face. We detail the attack vectors of Knative, and finally, we enumerate the threat actors impacting the Knative threat model.

Knative Eventing

Knative Eventing is a library used for developing applications on an event-driven architecture. A high-level goal of Knative Enting is to handle the transport of events from event producers to event consumers. Event producers and consumers are external to the Knative Eventing ecosystem - also called the Event Mesh. A producer is anything that can produce an event, such as external clients, applications, humans or IoT devices. A consumer is a service that receives the data from the event and processes it. This can for example be a cloud service, an application, a database or something else. The Knative Event Mesh is responsible for relaying the event from the producer to the consumer.

Knative Eventing consists of three main parts: Event Sources, a broker and triggers. Event Sources are the entrypoints into Knative Eventing and receive ingress traffic from the users choice of tooling. To illustrate this in practice, a user could write an Event Source to receive requests from their Slack workspace in their Knative Eventing deployment. In fact, a Slack Event Source has been suggested by the Knative community in the past[1]. Knative maintains a list of optional official Event Sources in the Knative-Extensions repository[2]. At the time of this audit, these are:

#	Name	Release status
1	Eventing-Autoscaler-Keda	Alpha
2	Eventing-Ceph	Beta
3	Eventing-CouchDB	Alpha
4	Eventing-Github	Alpha
5	Eventing-Gitlab	Alpha
6	Eventing-Istio	Beta
7	Eventing-Kafka	GA
8	Eventing-Kogito	Alpha
9	Eventing-NATS	Beta
10	Eventing-RabbitMQ	GA

[1] https://github.com/knative/eventing-contrib/issues/344
[2] https://github.com/knative-extensions

11	Eventing-Redis	Beta

The Broker and the Triggers handle the routing of incoming events. The Broker receives the event from the Event Source in the form of an HTTP request, parses it to a CloudEvents request and relays it to the trigger over a channel.

Below we cover the data and trust flow of the Knative Event Mesh.

Trustflow analysis

In this part of the Knative Eventing threat modelling we frame the data and trust flow of Knative Eventing. We illustrate this by way of diagram 1.0.0 that shows the data and trust flow of Knative eventing. At the top of the diagram are the event producers. These are exemplified by a github repository for the Eventing-Github Event Source, a Gitlab repository for the Eventing-Gitlab Event Source and a User. The User event producer demonstrates that Knative Eventing accepts calls directly to the broker. The event producers are encapsulated in a red box that denotes that these are untrusted entities. From the event producers to the Event Sources and further to the Broker, the trust flows low to high. From the Broker to the Triggers, the data flows with no change in level of trust, and finally the trust flows high to low from the Triggers to the event consumers, which in the diagram are exemplified as remote services - ie. a database, another cloud service, an application API or something else.

Trustflow overview

From component	To component	Level of trust flow
Event producers	Event Sources	Low to high
Event producers	Broker	Low to high
Event Sources	Broker	Low to high
Broker	Triggers	No change in trust
Triggers	Event consumers	High to low

Trustflow diagram

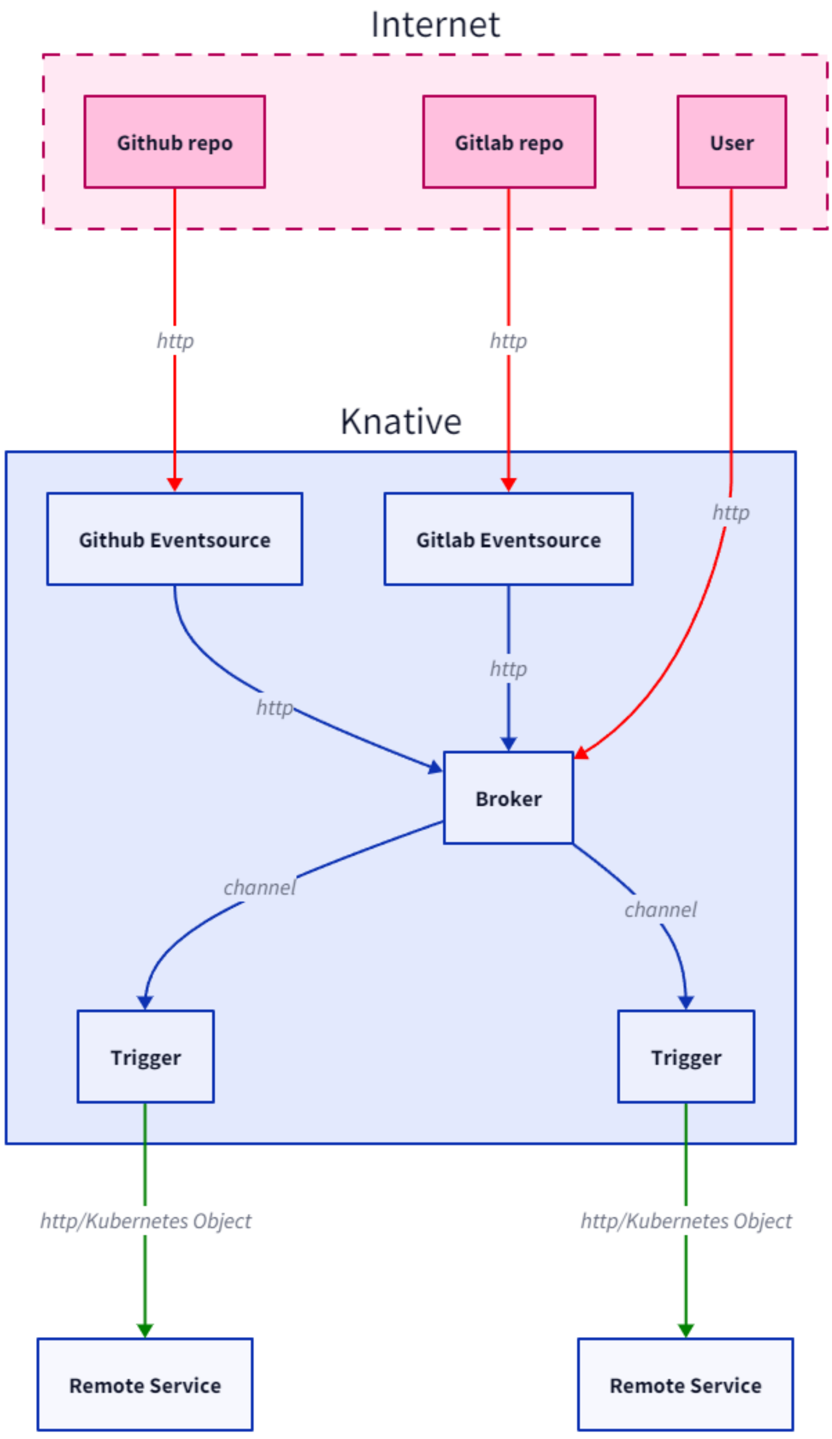


Figure 1.0.0: Trust and data flow of the Knative Event Mesh

Knative Eventing threat actors

In this section, we enumerate the threat actors of the Knative ecosystem. A threat actor is an individual or group that intentionally attempts to exploit vulnerabilities, deploy malicious code, or compromise or disrupt a Knative Eventing deployment, often for financial gain, espionage, or sabotage.
We identify the following threat actors below. For example, a fully untrusted user can also be a contributor to a 3rd-party library used by Knative Eventing. A threat actor can assume multiple profiles from the table.

Actor	**Description**	**Level of trust**
External attacker	Users that have not been granted any privileges and are unauthenticated.	Fully untrusted
Internal users	These are users that have permissions to navigate and modify the environment that produces the events. In the case of the Github Event Source, this would be users with access to the Github repository. In the case of an Event Source for Slack, this would be users in the Slack channel.	Limited trust
Internal admins	These are users who manage the environment that produces the events. In the case of the GitHub Event Source, these are admins of the GitHub repository. In the case of an Event Source for Slack, these are the admins of the Slack channel.	Limited trust
Cluster operator	A user with permissions to manage the Kubernetes cluster for deployments of Knative Eventing.	Fully trusted
Contributors to 3rd-party dependencies	Contributors to dependencies used by Knative Eventing.	Fully untrusted
Well-funded criminal groups	Organized criminal groups that often have either political or economic goals. These groups typically have large resources available and specific goals to achieve.	Fully untrusted

Knative Serving

From a high level, Knative Serving is an autoscaler. It manages the infrastructure to autoscale based on the amount of incoming traffic and the user's configuration. Knative Serving intercepts and evaluates traffic from the cluster or the internet before it reaches the user's application. Knative Serving will autoscale the necessary infrastructure based on the amount of traffic.

Users will deploy their applications in the Knative Service Pod. Knative runs a sidecar container called Queue-Proxy next to the user's application. Queue-Proxys job is to collect traffic metrics at runtime and impose the required concurrency of traffic to the users application container. Queue-Proxy can also queue traffic.

The autoscaler communicates to the Kubernetes Apiserver and sets the desired state of the cluster.

Trustflow analysis

Traffic enters Knative Serving through the ingress gateway. The ingress gateway is not the Kubernetes Ingress Gateway but rather an abstract representation of exposing the Knative infrastructure to the cluster. Knative can also expose the ingress gateway to traffic from outside the cluster by way of a Kubernetes `LoadBalancer` or `NodePort`. The ingress gateway is pluggable and does not have a standard implementation. From the ingress gateway, traffic flows to either the Activator or a Knative Service Pod, dependending on the user's configuration.

The traffic flows from the Activator to the Knative Service Pod. The Activator does not forward traffic to the autoscaler. Rather, the autoscaler probes the activator to scale up or down.

When traffic reaches the Knative Service Pod, it first flows through Queue-Proxy before it arrives at the destination: The user container. Users can optionally enable Security-Guard in the Queue-Proxy sidecar. Security-Guard is an official Knative extension that is not enabled by default in an out-of-the-box Knative deployment. It maintains a collection of micro-rules that Security-Guard uses to identify attempts to exploit a vulnerability in the user's application or its dependencies. The trust of the traffic from the ingress gateway and activator flows low to high to Security-Guard. After Security-Guard traffic flows with an unchanged level of trust to the user container. Note that the user application may need to do authentication or authorization of the request; however, from the perspective of Knatives security model, this is entirely the responsibility of the user's application.

There is a line of trust flow from the Kubernetes Apiserver once the autoscaler redefines the desired state of the cluster. This is in case a change happens to the image reference in the user container, and Kubernetes will fetch the image from the user-provided image reference. The registry is untrusted, and as such, data flows high to low from the K8s API Server to the registry and low to high from the registry to the API Server. While this is an attack surface for a Knative deployment, Knative relies on Kubernetes to fetch the correct image to the cluster and validate it.

From component	To component	Level of trust flow
Internet/cluster	Ingress gateway	Low to high
Ingress gateway	Activator	Low to low
Ingress gateway	Queue-Proxy	Low to high
Queue-Proxy	Security-Guard	Low to high
Security-Guard	User container	High to high

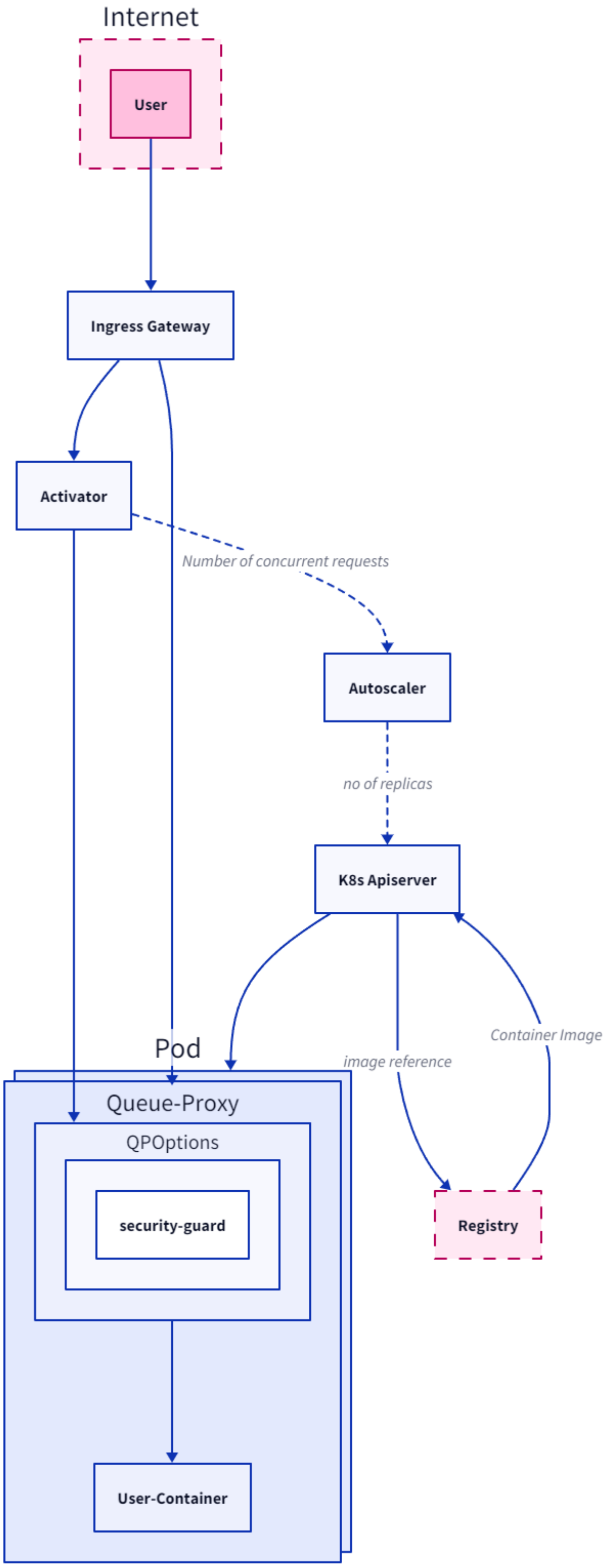


Figure 1.0.1: Trust and data flow of the Knative Serving

Knative Serving threat actors

A threat actor is an individual or group that intentionally attempts to exploit vulnerabilities, deploy malicious code, or compromise or disrupt a Knative deployment, often for financial gain, espionage, or sabotage. A threat actor is the personification of a possible attacker of security issues. Each threat actor has a level of trust tied to them, and matching one or several threat actors with Knative's threat model helps identify the high-level security risk. We identify the following threat actors for Knative. A threat actor can assume multiple profiles from the table below; for example, a fully untrusted user can also be a contributor to a 3rd-party library used by Knative.

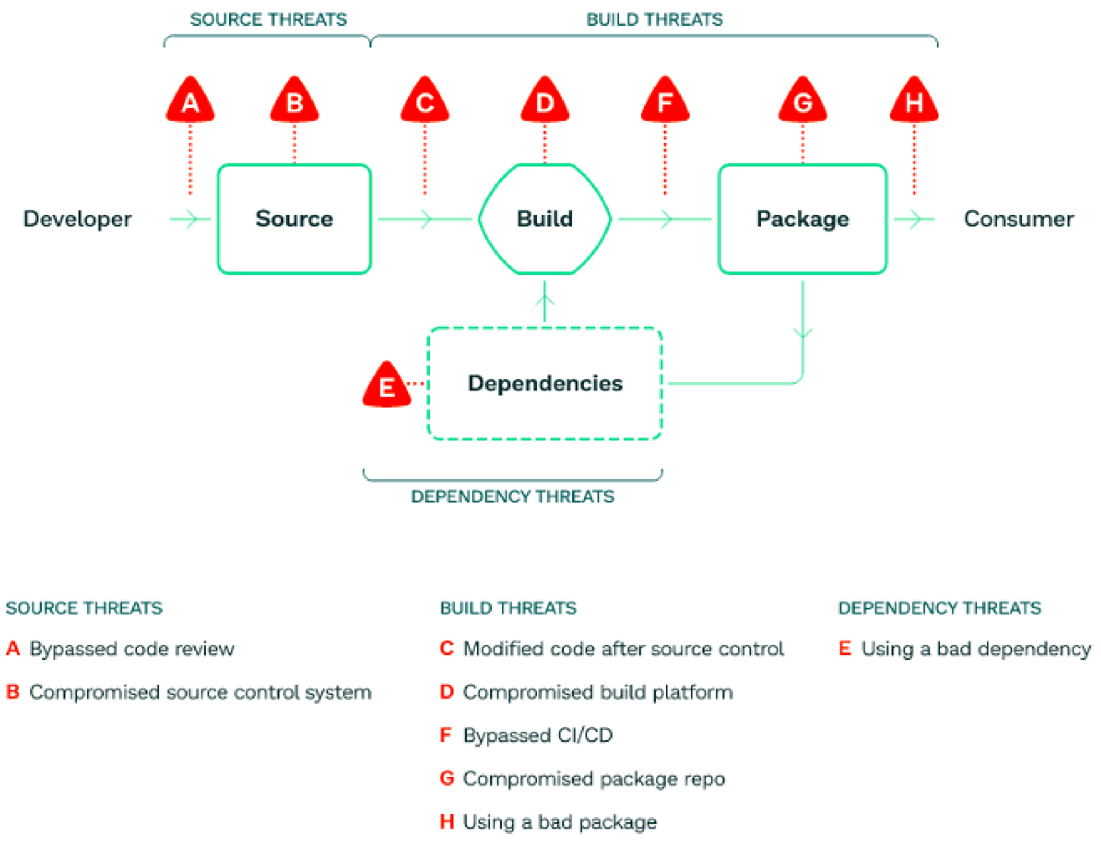
Actor	**Description**	**Level of trust**
External attacker	Users that have not been granted any privileges and are unauthenticated. If the Knative admin has exposed their Serving deployment to the internet, then this threat has the Ingress Gateway as their main legitimate attack surface. This threat actor is not a threat if Serving is not exposed to the internet.	Fully untrusted
Cluster operator	A user who has permissions to manage the Kubernetes cluster for deployments of Knative Serving.	Fully trusted
Contributors to Knative Serving	Code contributors to Knative Serving. This threat actor will compromise a Knative deployment by adding or finding vulnerabilities in 3rd-party libraries used by Knative Serving.	Fully untrusted
Contributors to 3rd-party dependencies	Contributors to dependencies used by Knative Serving. This threat actor will compromise a Knative Deployment by adding or finding vulnerabilities in 3rd-party libraries used by Knative Serving.	Fully untrusted
Malicious image maintainer	A threat actor that maintains an image in a public registry used by a Knative Serving user, and who deliberately publishes a malicious image.	Fully untrusted
Well-funded criminal groups	Organized criminal groups that often have either political or economic goals. These groups typically have large resources available and specific goals to achieve.	Fully untrusted

SLSA

ADA Logics conducted a Supply Chain Levels for Software Artifacts (SLSA) review of Knative. SLSA (https://github.com/slsa.dev) is a framework for assessing the security practices of a given software project with a focus on mitigating supply-chain risk. SLSA emphasises tamper resistance of artifacts as well as ephemerality of the build and release cycle.

SLSA mitigates a series of attack vectors in the software development life cycle (SDLC), all of which have seen real-world examples of successful attacks against open-source and proprietary software.

Below we see a diagram made by the SLSA illustrating the attack surface of the SDLC.



Each of the red markers show different areas of possible compromise that could allow attackers to tamper with the artifact that the consumer invokes at the end of the SDLC.

SLSA splits its assessment criteria into 4, increasingly demanding levels ranging from level 0 to 3. The higher the level of compliance, the higher tamper-resistance the project ensures its consumers.

Knative maintains its own generator to generate and include provenance in releases. The provenance generator is maintained at https://github.com/knative/toolbox/tree/main/provenance-generator and is a wrapper around Kubernetes' provenance generator. The provenance generator runs as part of the release on the line marked below:

https://github.com/knative/hack/blob/7030d5bf584de17a82ec8742c1078ca50633d45e/release.sh#L404-L421

```
  ID_TOKEN=$(gcloud auth print-identity-token --audiences=sigstore \
    --include-email \
    --impersonate-service-account="${SIGNING_IDENTITY}")
  echo "Signing Images with the identity ${SIGNING_IDENTITY}"
  ## Sign the images with cosign
  if [[ -f "$IMAGES_REFS_FILE" ]]; then
    COSIGN_EXPERIMENTAL=1 cosign sign $(cat "$IMAGES_REFS_FILE") \
      --recursive --identity-token="${ID_TOKEN}"
    cp "${IMAGES_REFS_FILE}" "${ARTIFACTS}"
    if  [ -n "${ATTEST_IMAGES:-}" ]; then # Temporary Feature Gate
      provenance-generator --clone-log=/logs/clone.json \
        --image-refs="$IMAGES_REFS_FILE" --output=attestation.json
      mkdir -p "${ARTIFACTS}" && cp attestation.json "${ARTIFACTS}"
      COSIGN_EXPERIMENTAL=1 cosign attest $(cat "$IMAGES_REFS_FILE") \
        --recursive --identity-token="${ID_TOKEN}" \
        --predicate=attestation.json --type=slsaprovenance
    fi
  fi
```

Consumers can download and verify provenance by following these steps:
First install cosign: https://docs.sigstore.dev/system_config/installation/

Next, download the provenance of the Knative v1.12.1 Eventing image, `gcr.io/knative-releases/knative.dev/eventing/cmd/controller@sha256:2ba4b01d6f3652818aa85f7fe0939e869b43734241f1f9bdda0f9db760e1dfa9, by running this command`:

```
cosign download attestation $(curl -L https://github.com/knative/eventing/releases/download/knative-v1.12.1/eventing-core.yaml | grep image: | head -1 | cut -d: -f2-)
```

To read the provenance, add the following to the above command: `| jq -r '.payload' | base64 --decode | jq`.

You should see the following provenance:

```
{
  "_type": "https://in-toto.io/Statement/v0.1",
  "predicateType": "https://slsa.dev/provenance/v0.2",
  "subject": [
    {
      "name": "gcr.io/knative-releases/knative.dev/eventing/cmd/controller",
      "digest": {
        "sha256": "2ba4b01d6f3652818aa85f7fe0939e869b43734241f1f9bdda0f9db760e1dfa9"
      }
    }
  ],
  "predicate": {
    "builder": {
      "id": "https://prow.knative.dev"
    },
    "buildType": "https://prow.knative.dev/ProwJob@v1",
    "invocation": {
      "configSource": {
        "entryPoint":
"https://github.com/knative/test-infra/tree/main/prow/jobs/generated/knative"
      }
    },
    "buildConfig": {
      "command": [
        "runner.sh",
        "./hack/release.sh",
        "--dot-release",
        "--release-gcs",
        "knative-releases/eventing",
        "--release-gcr",
        "gcr.io/knative-releases",
        "--github-token",
        "/etc/hub-token/token",
        "--branch",
        "release-1.12"
      ],
      "entrypoint": {
        "args": [
          "runner.sh",
          "./hack/release.sh",
          "--dot-release",
          "--release-gcs",
          "knative-releases/eventing",
          "--release-gcr",
          "gcr.io/knative-releases",
          "--github-token",
          "/etc/hub-token/token",
          "--branch",
          "release-1.12"
        ],
        "artifact_dir": "/logs/artifacts",
        "container_name": "test",
        "grace_period": 15000000000,
        "marker_file": "/logs/marker-file.txt",
        "metadata_file": "/logs/artifacts/metadata.json",
        "process_log": "/logs/process-log.txt",
        "timeout": 14400000000000
      },
      "prowjob": {
        "metadata": {
          "annotations": {
            "prow.k8s.io/context": "",
            "prow.k8s.io/job": "release_eventing_release-1.12_periodic",
            "testgrid-dashboards": "knative-release-1.12",
            "testgrid-tab-name": "eventing-release"
          },
          "creationTimestamp": "2023-11-28T09:38:15Z",
          "generation": 4,
          "labels": {
            "created-by-prow": "true",
            "prow.k8s.io/build-id": "1729434521369055232",
            "prow.k8s.io/context": "",
            "prow.k8s.io/id": "88d5cb2c-0368-4cf1-9763-4917942e5dfd",
            "prow.k8s.io/job": "release_eventing_release-1.12_periodic",
            "prow.k8s.io/refs.base_ref": "release-1.12",
            "prow.k8s.io/refs.org": "knative",
            "prow.k8s.io/refs.repo": "eventing",
            "prow.k8s.io/type": "periodic"
          },
          "name": "88d5cb2c-0368-4cf1-9763-4917942e5dfd",
          "namespace": "default",
          "resourceVersion": "511241023",
          "uid": "3a7125a0-935b-4433-a30b-0430ae2b46e5"
        },
        "spec": {
          "agent": "kubernetes",
          "cluster": "prow-build",
          "decoration_config": {
            "gcs_configuration": {
              "bucket": "knative-prow",
              "path_strategy": "explicit"
            },
            "gcs_credentials_secret": "gcs-upload",
            "grace_period": "15s",
            "resources": {
              "sidecar": {
                "requests": {
                  "cpu": "100m",
                  "memory": "20Mi"
                }
              }
            },
"timeout": "4h0m0s",
            "utility_images": {
              "clonerefs": "gcr.io/k8s-prow/clonerefs:v20231121-4e39ac27ea",
              "entrypoint": "gcr.io/k8s-prow/entrypoint:v20231121-4e39ac27ea",
              "initupload": "gcr.io/k8s-prow/initupload:v20231121-4e39ac27ea",
              "sidecar": "gcr.io/k8s-prow/sidecar:v20231121-4e39ac27ea"
            }
          },
          "extra_refs": [
            {
              "base_ref": "release-1.12",
              "org": "knative",
              "path_alias": "knative.dev/eventing",
              "repo": "eventing"
            }
          ],
          "job": "release_eventing_release-1.12_periodic",
          "max_concurrency": 1,
          "namespace": "test-pods",
          "pod_spec": {
            "containers": [
              {
                "command": [
                  "runner.sh",
                  "./hack/release.sh",
                  "--dot-release",
                  "--release-gcs",
                  "knative-releases/eventing",
                  "--release-gcr",
                  "gcr.io/knative-releases",
                  "--github-token",
                  "/etc/hub-token/token",
                  "--branch",
                  "release-1.12"
                ],
                "env": [
                  {
                    "name": "GOOGLE_APPLICATION_CREDENTIALS",
                    "value": "/etc/release-account/service-account.json"
                  },
                  {
                    "name": "E2E_CLUSTER_REGION",
                    "value": "us-central1"
                  },
                  {
                    "name": "SIGN_IMAGES",
                    "value": "true"
                  },
                  {
                    "name": "ATTEST_IMAGES",
                    "value": "true"
                  }
                ],
                "image":
"us-docker.pkg.dev/knative-tests/images/prow-tests:v20231122-b38b92934",
                "name": "",
                "resources": {
                  "limits": {
                    "memory": "16Gi"
                  },
                  "requests": {
                    "memory": "12Gi"
                  }
                },
                "securityContext": {
                  "privileged": true
                },
                "volumeMounts": [
                  {
```

```
                  "mountPath": "/etc/hub-token",
                  "name": "hub-token",
                  "readOnly": true
                },
                {
                  "mountPath": "/etc/release-account",
                  "name": "release-account",
                  "readOnly": true
                }
              ]
            }
          ],
          "nodeSelector": {
            "kubernetes.io/arch": "amd64",
            "type": "testing"
          },
          "volumes": [
            {
              "name": "hub-token",
              "secret": {
                "items": [
                  {
                    "key": "hub_token",
                    "path": "token"
                  }
                ],
                "secretName": "github-credentials"
              }
            },
            {
              "name": "release-account",
              "secret": {
                "items": [
                  {
                    "key": "release.json",
                    "path": "service-account.json"
                  }
                ],
                "secretName": "prow-google-credentials"
              }
            }
          ]
        },
        "prowjob_defaults": {
          "tenant_id": "GlobalDefaultID"
        },
        "report": true,
        "type": "periodic"
      },
      "status": {
        "startTime": null
      }
    }
  },
  "metadata": {
    "buildInvocationID": "1729434521369055232",
      "buildStartedOn": "2023-11-28T09:38:15Z",
      "buildFinishedOn": "2023-11-28T12:01:52.247419694Z",
      "completeness": {
        "parameters": true,
        "environment": true,
        "materials": true
      },
      "reproducible": false
    },
    "materials": [
      {
        "uri": "git+https://github.com/knative/eventing",
        "digest": {
          "sha1": "747a9f54a6caa7c345f0b605520aecf85908bf49"
        }
      }
    ]
  }
}
```

This is a SLSA-compliant provenance. Note that it is SLSA v0.2 whereas the latest SLSA release is v1.0. Consumers can verify the provenance by following the steps outlined here: https://slsa.dev/spec/v1.0/verifying-artifacts.

During the audit, the Knative Maintainers found that Knative Serving was not including provenance. This was due to a few missing lines in the Prow configuration, and the Knative maintainers fixed that: https://github.com/knative/infra/pull/288.

Issues found

#	ID	Title	Severity	Fixed
1	ADA-KNATIVE-23-1	Issue in third-party dependency	Moderate	In progress
2	ADA-KNATIVE-23-2	3rd-party dependency uses insecure cryptographic primitive for sensitive data	Informational	In progress
3	ADA-KNATIVE-23-3	Slice bound out of range in 3rd-party dependency	Informational	In progress
4	ADA-KNATIVE-23-4	Two potential slowloris attacks in eventing-github	Low	Yes
5	ADA-KNATIVE-23-5	Security Guard exposes profiling endpoints by default	Low	Yes
6	ADA-KNATIVE-23-6	Two potential slowloris attacks in Security Guard	Low	Yes
7	ADA-KNATIVE-23-7	Remote code execution from lack of image validation in Knative Func	High	Yes
8	ADA-KNATIVE-23-8	Lack of logging in case image is referenced by tag	Moderate	Yes
9	ADA-KNATIVE-23-9	Possible infinity loop over untrusted image	Moderate	Yes
10	ADA-KNATIVE-23-10	Attacker-controlled pod can cause denial of service of autoscaler	Moderate	Yes
11	ADA-KNATIVE-23-11	Out of bounds read panic in Security-guard authentication	Informational	Yes
12	ADA-KNATIVE-23-12	Missing SECURITY.md file	Informational	Yes
13	ADA-KNATIVE-23-13	Possible DoS in Security Guard /sync endpoint	Moderate	Yes
14	ADA-KNATIVE-23-14	Possible DoS in Security Guard /mutate endpoint	Moderate	Yes
15	ADA-KNATIVE-23-15	Potential slowloris attack in Eventing-Gitlab	Low	Yes
16	ADA-KNATIVE-23-16	Hard-coded insecure protocol used by Knative Serving Activator	Low	No

Issue in third-party dependency

ID	ADA-KNATIVE-23-1
Component	Eventing-Github
Severity	Moderate
Status: Fix in progress	

Ada Logics found an issue in a third-party dependency, which is currently being triaged by the dependency maintainers. Ada Logics have submitted a fix that is pending a merge.

3rd-party dependency uses insecure cryptographic primitive for sensitive data

ID	ADA-KNATIVE-23-2
Component	Eventing-Github
Severity	Informational
Status: Fixed in https://github.com/go-playground/webhooks/pull/173, not yet released	

Eventing-Github's handler extracts the payload of incoming requests using the `github.com/go-playground/webhooks/v5/webhooks/github.(Webhook).Parse` API. This parsing routine uses SHA1 to verify incoming signatures against the hooks secret:

https://github.com/go-playground/webhooks/blob/659b2a276b2274719c30d765f4328ed340f01904/github/github.go#L162-L174

```
        if len(hook.secret) > 0 {
                signature := r.Header.Get("X-Hub-Signature")
                if len(signature) == 0 {
                        return nil, ErrMissingHubSignatureHeader
                }
                mac := hmac.New(sha1.New, []byte(hook.secret))
                _, _ = mac.Write(payload)
                expectedMAC := hex.EncodeToString(mac.Sum(nil))

                if !hmac.Equal([]byte(signature[5:]), []byte(expectedMAC)) {
                        return nil, ErrHMACVerificationFailed
                }
        }
```

SHA1 is broken for some use cases and NIST has declared that it should be fully phased out by 2030[3]. The impact of using SHA1 in this scenario is low but does not represent best practices.

SHA256 version is available:
https://docs.github.com/en/webhooks-and-events/webhooks/securing-your-webhooks#validating-payloads-from-github

A third-party contributor has already made a PR for this issue which is pending merge:
https://github.com/go-playground/webhooks/pull/173

[3] https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm

Slice bound out of range in 3rd-party dependency

ID	ADA-KNATIVE-23-3
Component	Eventing-Github
Severity	Informational
Status: Fixed in https://github.com/go-playground/webhooks/pull/173, not yet released	

Eventing-Github's handler extracts the payload of incoming requests using the `github.com/go-playground/webhooks/v5/webhooks/github.(Webhook).Parse` API. This parsing routine has a slice bounds out of range from reading a string signature without checking its length first:

https://github.com/go-playground/webhooks/blob/659b2a276b2274719c30d765f4328ed340f01904/github/github.go#L162-L174

```go
        if len(hook.secret) > 0 {
                signature := r.Header.Get("X-Hub-Signature")
                if len(signature) == 0 {
                        return nil, ErrMissingHubSignatureHeader
                }
                mac := hmac.New(sha1.New, []byte(hook.secret))
                _, _ = mac.Write(payload)
                expectedMAC := hex.EncodeToString(mac.Sum(nil))

                if !hmac.Equal([]byte(signature[5:]), []byte(expectedMAC)) {
                        return nil, ErrHMACVerificationFailed
                }
        }
```

This is a recoverable issue with limited impact.

Two potential slowloris attacks in eventing-github

ID	ADA-KNATIVE-23-4
Component	Eventing-Github
Severity	Low
Status: Fixed	

Slowloris is a type of attack where an attacker opens a connection between their controlled machine and the victim's server. Once the attacker has opened the connection, they keep it open for as long as possible. They will do the same with a large number of controlled machines to hog the available connections and prevent other users from accessing the service. As such, the victim's server stays up but remains busy from processing the attacker's requests and becomes unavailable to legitimate users.

An attacker can exploit a Slowloris issue by identifying execution paths in their target application that cause it to take longer time to return from, and the attacker can then send requests that force the application into these. The fact that the Eventing-Github server is susceptible to a Slowloris attack does not mean that it is easily exploitable.

The following servers do not set a `ReadHeaderTimeout` which could lead do a DDoS attack, where a large group of users send requests to the server causing the server to hang for long enough to deny it from being available to other users, also known as a Slowloris attack:

https://github.com/knative-extensions/eventing-github/blob/2e12b307bb8905dbc8dad8dedc475e5f34ef8efb/pkg/mtadapter/adapter.go#L89-L91

```
	server := &http.Server{
		Addr:    fmt.Sprintf(":%d", a.port),
		Handler: a.router,
	}
```

https://github.com/knative-extensions/eventing-github/blob/9c53cef7fa9a884d65523772216e9fe870d7663f/pkg/adapter/adapter.go#L81-L84

```
	server := &http.Server{
		Addr:    ":" + a.port,
		Handler: a.newRouter(),
	}
```

An attacker needs a way to cause eventing-github to run slowly such that multiple invocations would generate a queue of pending requests. The fact that the `ReadHeaderTimeout` is not set does not mean that a Slowloris attack is possible, however, even if an attacker is not able to cause eventing-github to run slowly, we advise that `ReadHeaderTimeout` be added to guard against any Slowloris attacks in the future.

Security Guard exposes profiling endpoints by default

ID	ADA-KNATIVE-23-5
Component	Security Guard
Severity	Low
Status: Fixed	

Exposed profiling endpoints may reveal sensitive data to attackers that are in a position to access them. Profiling endpoints should not be enabled by default; rather, they should be exposed if the user specifically enables them. Security Guard has two cases where profiling endpoints are enabled by default.

https://github.com/knative-extensions/security-guard/blob/76f34f56713ca88c6813d732ff2e27938f0d5195/pkg/iodup/iodup.go#L22

```
import (
        "fmt"
        "io"
        _ "net/http/pprof"
        "sync"
        "time"
)
```

https://github.com/knative-extensions/security-guard/blob/76f34f56713ca88c6813d732ff2e27938f0d5195/pkg/guard-gate/gate.go#L28

```
import (
        "context"
        "errors"
        "net/http"
        "os"
        "regexp"
        "strings"
        "time"

        _ "net/http/pprof"

        pi "knative.dev/security-guard/pkg/pluginterfaces"
)
```

Two potential slowloris attacks in Security Guard

ID	ADA-KNATIVE-23-6
Component	Security Guard
Severity	Low
Status: Fixed	

Slowloris is a type of attack where an attacker opens a connection between their controlled machine and the victim's server. Once the attacker has opened the connection, they keep it open for as long as possible. They will do the same with a large number of controlled machines to hog the available connections and prevent other users from accessing the service. As such, the victim's server stays up but remains busy from processing the attacker's requests and becomes unavailable to legitimate users.

An attacker can exploit a Slowloris issue by identifying execution paths in their target application that cause it to take longer time to return from, and the attacker can then send requests that force the application into these. The fact that the Security-Guard server is susceptible to a Slowloris attack does not mean that it is easily exploitable.

The following servers do not set a `ReadHeaderTimeout`, which could lead do a DDoS attack, where a large group of users send requests to the server, causing the server to hang for long enough to deny it from being available to other users, also know as a Slowloris attack:

https://github.com/knative-extensions/security-guard/blob/f3303bbf61dc85eb5ad7a6033e0a1b319f10a45d/cmd/guard-webhook/main.go#L269-L275

```
	server := &http.Server{
		Handler: mux,
		Addr:    ":8443",
		TLSConfig: &tls.Config{
			Certificates: []tls.Certificate{serverCert},
		},
	}
```

https://github.com/knative-extensions/security-guard/blob/f3303bbf61dc85eb5ad7a6033e0a1b319f10a45d/cmd/guard-rproxy/main.go#L202-L205

```
	srv := &http.Server{
		Addr:    target,
		Handler: mux,
	}
```

An attacker needs a way to cause Security Guard to run slowly such that multiple invocations would generate a queue of pending requests. The fact that the `ReadHeaderTimeout` is not set does not mean that a Slowloris attack is possible; however, even if an attacker is not able to cause security-guard to run slowly, we advise that `ReadHeaderTimeout` be added to defend against any Slowloris attacks in the future.

Remote code execution from lack of image validation in Knative Func

ID	ADA-KNATIVE-23-7
Component	Knative Func
Severity	High
Status: Fixed	

When Knative Func pulls an image to get its config file, Knative does not validate the fetched image and will not detect any potential tampering.

This issue allows a malicious threat actor to deliver a malicious image to the Knative Func user in `knative.dev/func/pkg/builders/s2i.s2iScriptURL` which extracts the `io.openshift.s2i.scripts-url` label of an image and passes them onto the builder. Labels are optional pieces of metadata about a container image. The `io.openshift.s2i.scripts-url` label is typically used to run *assemble* and *run*[4] scripts for S2I builder images. The `io.openshift.s2i.scripts-url` points to a directory which contains executable scripts used for packaging and running an artifact. These are typically an *assemble* script and a *run* script; The *assemble* script builds the applications artifacts and the *run* script runs the application. They can be implemented in any programming language that allows them to be executable in the S2I builder image.

To exploit this vulnerability in Knative Func, an attacker needs to control the registry that the image reference points to, and they need to be able to return a malicious image to the Knative Func user. They could achieve this position for example by compromising a user account of the image on the registry, by overtaking a forgotten user on the registry, achieving admin status of the image on the registry by asking for it or by compromising the registry. With this position, the attacker is able to return their own malicious image to the Knative Func user. The attacker will craft an image with the label `"io.openshift.s2i.scripts-url"` with a URL to malicious scripts that the attacker wants to execute in the victim's Dockerfile. This URL can be local or remote. When the Knative Func user builds a function using the S2I builder, the workflow proceeds as follows.

The attacker delivers a malicious image with a config file containing a URL to a malicious script. Below, the attacker controls the `cfg` in a successful attack.

[4] https://docs.openshift.com/container-platform/3.11/creating_images/s2i.html#s2i-scripts

https://github.com/knative/func/blob/c15450177a44aa98f8ccc50d0a787b01594ce915/pkg/builders/s2i/builder.go#L359

```
func s2iScriptURL(ctx context.Context, cli DockerClient, image string) (string, error) {
        img, _, err := cli.ImageInspectWithRaw(ctx, image)
        if err != nil {
                if dockerClient.IsErrNotFound(err) { // image is not in the daemon, get 
info directly from registry
                        var (
                                ref name.Reference
                                img v1.Image
                                cfg *v1.ConfigFile
                        )

                        ref, err = name.ParseReference(image)
                        if err != nil {
                                return "", fmt.Errorf("cannot parse image name: %w", err)
                        }
                        img, err = remote.Image(ref)
                        if err != nil {
                                return "", fmt.Errorf("cannot get image from registry: %w", 
err)
                        }
                        cfg, err = img.ConfigFile()
                        if err != nil {
                                return "", fmt.Errorf("cannot get config for image: %w", 
err)
                        }

                        if cfg.Config.Labels != nil {
                                if u, ok := 
cfg.Config.Labels["io.openshift.s2i.scripts-url"]; ok {
                                        return u, nil
                                }
                        }
                }
                return "", err
        }

        if img.Config != nil && img.Config.Labels != nil {
                if u, ok := img.Config.Labels["io.openshift.s2i.scripts-url"]; ok {
                        return u, nil
                }
        }

        if img.ContainerConfig != nil && img.ContainerConfig.Labels != nil {
                if u, ok := img.ContainerConfig.Labels["io.openshift.s2i.scripts-url"]; ok 
{
                        return u, nil
                }
        }

        return "", nil
}
```

The value of `io.openshift.s2i.scripts-url` gets stored in the build `cfg`:

https://github.com/knative/func/blob/c15450177a44aa98f8ccc50d0a787b01594ce915/pkg/builders/s2i/builder.go#L187

```
        scriptURL, err := s2iScriptURL(ctx, client, cfg.BuilderImage)
        if err != nil {
                return fmt.Errorf("cannot get s2i script url: %w", err)
        }
        cfg.ScriptsURL = scriptURL
```

The S2I builder builds using the parameter:

https://github.com/knative/func/blob/c15450177a44aa98f8ccc50d0a787b01594ce915/pkg/builders/s2i/builder.go#L229-L232

```
        result, err := impl.Build(cfg)
        if err != nil {
                return
        }
```

In the case of the Dockerfile strategy implementation, S2I will fetch the attacker-provided scripts:

https://github.com/openshift/source-to-image/blob/980ca195116928b3beb61b25d5939d0044b3040b/pkg/build/strategies/dockerfile/dockerfile.go#L352

```
        // Install scripts provided by user, overriding all others.
        // This _could_ be an image:// URL, which would override any scripts above.
        urlScripts := builder.installScripts(config.ScriptsURL, config)
```

`installScripts` creates a new installer and installs the scripts:

https://github.com/openshift/source-to-image/blob/980ca195116928b3beb61b25d5939d0044b3040b/pkg/build/strategies/dockerfile/dockerfile.go#L395-L409

```
func (builder *Dockerfile) installScripts(scriptsURL string, config *api.Config)
[]api.InstallResult {
        scriptInstaller := scripts.NewInstaller(
                "",
                scriptsURL,
                config.ScriptDownloadProxyConfig,
                nil,
                api.AuthConfig{},
                builder.fs,
                config,
        )

        // all scripts are optional, we trust the image contains scripts if we don't find
them
        // in the source repo.
        return scriptInstaller.InstallOptional(append(scripts.RequiredScripts,
scripts.OptionalScripts...), config.WorkingDir)
}
```

The scripts installer has different implementations based on the type of scripts. One of these is the `URLScriptHandler` which downloads the scripts from a URL:

https://github.com/openshift/source-to-image/blob/980ca195116928b3beb61b25d5939d0044b3040b/pkg/scripts/install.go#L74-L111

```
func (s *URLScriptHandler) Get(script string) *api.InstallResult {
        if len(s.URL) == 0 {
                return nil
        }
        scriptURL, err := url.ParseRequestURI(s.URL + "/" + script)
        if err != nil {
                log.Infof("invalid script url %q: %v", s.URL, err)
                return nil
        }
        return &api.InstallResult{
                Script: script,
                URL:    scriptURL.String(),
        }
}

// Install downloads the script and fix its permissions.
func (s *URLScriptHandler) Install(r *api.InstallResult) error {
        downloadURL, err := url.Parse(r.URL)
        if err != nil {
                return err
        }
        dst := filepath.Join(s.DestinationDir, constants.UploadScripts, r.Script)
        if _, err := s.Download.Download(downloadURL, dst); err != nil {
                if e, ok := err.(s2ierr.Error); ok {
                        if e.ErrorCode == s2ierr.ScriptsInsideImageError {
                                r.Installed = true
                                return nil
                        }
                }
                return err
        }
        if err := s.FS.Chmod(dst, 0755); err != nil {
                return err
        }
        r.Installed = true
        r.Downloaded = true
        return nil
}
```

Lack of logging in case image is referenced by tag

ID	ADA-KNATIVE-23-8
Component	Knative Func
Severity	Moderate
Status: Fixed	

Knative Func does not log a warning in case a user references an image by tag. Referencing by tag is the worse practice of the two options and increases the likelihood for a threat actor to tamper with the image:

https://github.com/knative/func/blob/c15450177a44aa98f8ccc50d0a787b01594ce915/pkg/builders/s2i/builder.go#L345-L390

```
func s2iScriptURL(ctx context.Context, cli DockerClient, image string) (string, error) {
        img, _, err := cli.ImageInspectWithRaw(ctx, image)
        if err != nil {
                if dockerClient.IsErrNotFound(err) { // image is not in the daemon, get
info directly from registry
                        var (
                                ref name.Reference
                                img v1.Image
                                cfg *v1.ConfigFile
                        )

                        ref, err = name.ParseReference(image)
                        if err != nil {
                                return "", fmt.Errorf("cannot parse image name: %w", err)
                        }
                        img, err = remote.Image(ref)
                        if err != nil {
                                return "", fmt.Errorf("cannot get image from registry: %w",
err)
                        }
                        cfg, err = img.ConfigFile()
                        if err != nil {
                                return "", fmt.Errorf("cannot get config for image: %w",
err)
                        }

                        if cfg.Config.Labels != nil {
                                if u, ok :=
cfg.Config.Labels["io.openshift.s2i.scripts-url"]; ok {
                                        return u, nil
                                }
                        }
                }
                return "", err
        }
        if img.Config != nil && img.Config.Labels != nil {
                if u, ok := img.Config.Labels["io.openshift.s2i.scripts-url"]; ok {
                        return u, nil
                }
        }

        if img.ContainerConfig != nil && img.ContainerConfig.Labels != nil {
                if u, ok := img.ContainerConfig.Labels["io.openshift.s2i.scripts-url"]; ok
{
                        return u, nil
                }
        }

        return "", nil
}
```

Possible infinity loop over untrusted image

ID	ADA-KNATIVE-23-9
Component	Knative Func
Severity	Moderate
Status: Fixed	

Knative Func loops over the index manifests of an image coming from the registry without enforcing a limit to the number of manifests. This could allow a malicious image to cause an infinite loop in Knative Func with a high number of manifests. To utilize this vulnerability, the attacker needs to control the registry from which Knative Func fetches the image or be able to control the response in another way when Knative Func sends the request to the registry.

https://github.com/knative/func/blob/5a4803bf959852737a25ed558dcae891b80ab30f/pkg/docker/platform.go#L63

```go
func GetPlatformImage(ref, platform string) (string, error) {

        plat, err := platforms.Parse(platform)
        if err != nil {
                return "", fmt.Errorf("cannot parse platform: %w", err)
        }

        r, err := name.ParseReference(ref)
        if err != nil {
                return "", fmt.Errorf("cannot parse reference: %w", err)
        }

        desc, err := remote.Get(r)
        if err != nil {
                return "", fmt.Errorf("cannot get remote image: %w", err)
        }

        if desc.MediaType != gcrTypes.OCIImageIndex && desc.MediaType !=
gcrTypes.DockerManifestList {
                // it's non-multi-arch image
                var img v1.Image
                var cfg *v1.ConfigFile
                img, err = desc.Image()
                if err != nil {
                        return "", fmt.Errorf("cannot get image from the descriptor: %w",
err)
                }
                cfg, err = img.ConfigFile()
                if err != nil {
                        return "", fmt.Errorf("cannot get config file for the image: %w",
err)
                }
                if plat.OS == cfg.OS &&
                        plat.Architecture == cfg.Architecture {
                        return ref, nil
                }
                return "", fmt.Errorf("the %q platform is not supported by the %q image",
platform, ref)
        }

        idx, err := desc.ImageIndex()
        if err != nil {
                return "", fmt.Errorf("cannot get image index: %w", err)
        }

        idxMft, err := idx.IndexManifest()
        if err != nil {
                return "", fmt.Errorf("cannot get index manifest: %w", err)
        }

        for _, manifest := range idxMft.Manifests {
                if plat.OS == manifest.Platform.OS &&
                        plat.Architecture == manifest.Platform.Architecture {
                        return r.Context().Name() + "@" + manifest.Digest.String(), nil
                }
        }

        return "", fmt.Errorf("the %q platform is not supported by the %q image",
platform, ref)
}
```

Attacker-controlled pod can cause denial of service of autoscaler

ID	ADA-KNATIVE-23-10
Component	Knative Serving
Severity	Moderate
Status: Fixed	

An attacker who controls a pod to a degree where they can control the responses from the `/metrics` endpoint can cause Denial-of-Service of the autoscaler from an unbound memory allocation bug. When the autoscaler scrapes the metrics of pods, it sends a request to the `/metrics` endpoint of each pod and reads the response entirely into memory. The root cause is in the `httpScrapeClient`, which parses the response from the pod into a `Stat` type:

https://github.com/knative/serving/blob/45f7c054f69448695d4e9bc11f5a451b3c9f1eff/pkg/autoscaler/metrics/http_scrape_client.go#L54-L71

```
func (c *httpScrapeClient) Do(req *http.Request) (Stat, error) {
        req.Header.Add("Accept", netheader.ProtobufMIMEType)
        resp, err := c.httpClient.Do(req)
        if err != nil {
                return emptyStat, err
        }
        defer resp.Body.Close()
        if resp.StatusCode < http.StatusOK || resp.StatusCode >=
http.StatusMultipleChoices {
                return emptyStat, scrapeError{
                        error:       fmt.Errorf("GET request for URL %q returned HTTP
status %v", req.URL.String(), resp.StatusCode),
                        mightBeMesh: nethttp.IsPotentialMeshErrorResponse(resp),
                }
        }
        if resp.Header.Get("Content-Type") != netheader.ProtobufMIMEType {
                return emptyStat, errUnsupportedMetricType
        }
        return statFromProto(resp.Body)
}
```

https://github.com/knative/serving/blob/45f7c054f69448695d4e9bc11f5a451b3c9f1eff/pkg/autoscaler/metrics/http_scrape_client.go#L80C6-L94

```
func statFromProto(body io.Reader) (Stat, error) {
        var stat Stat
        b := pool.Get().(*bytes.Buffer)
        b.Reset()
        defer pool.Put(b)
        _, err := b.ReadFrom(body)
        if err != nil {
                return emptyStat, fmt.Errorf("reading body failed: %w", err)
        }
        err = stat.Unmarshal(b.Bytes())
        if err != nil {
                return emptyStat, fmt.Errorf("unmarshalling failed: %w", err)
        }
        return stat, nil
}
```

During that parsing routine, Knative Serving will first read the body of the response into a buffer and then read the buffer into memory.

This is illustrated by adding the following unit test to `pkg/autoscaler/metrics/http_scrape_client_test.go`:

POC

```
func TestStats(t *testing.T) {
        b := bytes.Repeat([]byte("1337"), 1000000000)
        r1 := bytes.NewReader(b)
        r2 := bytes.NewReader(b)
        mr := io.MultiReader(r1, r2)
        statFromProto(mr)
}
```

This unit test will perform a sig kill with a temporary, machine-wide denial of service. On an 8-core machine, the machine freezes for around 20-30 seconds before Go performs a SigKill.

[WARNING: SAVE ALL WORK BEFORE REPRODUCING]
To test out the reproducer, run:

```
go test -run=TestStats
```

Now observe the memory usage and wait for the following stacktrace:

```
signal: killed
FAIL    knative.dev/serving/pkg/autoscaler/metrics      69.719s
```

Out of bounds read panic in Security-guard authentication

ID	ADA-KNATIVE-23-11
Component	Knative Security Guard
Severity	Informational
Status: Fixed	

Security-guards guard-services `baseHandler` reads the user's token when authenticating the request:

`baseHandler` is invoked as part of the handler for the `sync` endpoint - `processSync` - of the Security Guard `learner`:

https://github.com/knative-extensions/security-guard/blob/39559b7f81b973dd34ebb335b84053bd547cf5c1/cmd/guard-service/main.go#L332-L344

```
func (l *learner) init() (srv *http.Server, quit chan bool, flushed chan bool) {
	l.tokens = make(map[string]*tokenData)

	l.pileLearnTicker = utils.NewTicker(time.Second)
	l.pileLearnTicker.Start()

	l.chacheTokenTicker = utils.NewTicker(time.Minute * 10)
	l.chacheTokenTicker.Start()

	l.services = newServices()

	mux := http.NewServeMux()
	mux.HandleFunc("/sync", l.processSync)
```

`processSync` invokes `baseHandler` to retrieve a record and the pod name:

https://github.com/knative-extensions/security-guard/blob/39559b7f81b973dd34ebb335b84053bd547cf5c1/cmd/guard-service/main.go#L236C1-L243C3

```
func (l *learner) processSync(w http.ResponseWriter, req *http.Request) {
	var syncReq spec.SyncMessageReq
	var syncResp spec.SyncMessageResp

	record, podname, err := l.baseHandler(w, req)
	if err != nil {
		return
	}
```

`baseHandler` first authenticates the request before retrieving the record and pod name. Below, the highlighted line shows where `baseHandler` authenticates the request:

https://github.com/knative-extensions/security-guard/blob/39559b7f81b973dd34ebb335b84053bd547cf5c1/cmd/guard-service/main.go#L197-L234

```go
func (l *learner) baseHandler(w http.ResponseWriter, req *http.Request) (record
*serviceRecord, podname string, err error) {
        var sid, ns string
        var cmFlag bool

        if l.env.GuardServiceAuth != "false" {
                cmFlag, err = l.queryDataAuth(req.URL.Query())
                if err != nil {
                        pi.Log.Infof("queryData failed with %v", err)
                        http.Error(w, err.Error(), http.StatusBadRequest)
                        return
                }

                podname, sid, ns, err = l.authenticate(req)
                if err != nil {
                        pi.Log.Infof("authenticate failed with %v", err)
                        http.Error(w, err.Error(), http.StatusUnauthorized)
                        return
                }
        } else {
                cmFlag, podname, sid, ns, err = l.queryDataNoAuth(req.URL.Query())
                if err != nil {
                        pi.Log.Infof("queryData failed with %v", err)
                        http.Error(w, err.Error(), http.StatusBadRequest)
                        return
                }
        }

        // get session record, create one if does not exist
        record = l.services.get(ns, sid, cmFlag)
        if record == nil {
                // should never happen
                err = fmt.Errorf("no record created")
                pi.Log.Infof("internal error %v for request ns %s, sid %s, pod %s, cmFlag
%t", err, ns, sid, podname, cmFlag)
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
        return
}
```

When authenticating the request, Security Guard gets a token from a header of the request and reads it from index 7 on the line below.

https://github.com/knative-extensions/security-guard/blob/39559b7f81b973dd34ebb335b84053bd547cf5c1/cmd/guard-service/main.go#L104-L115

```go
func (l *learner) authenticate(req *http.Request) (podname string, sid string, ns 
string, err error) {
        token := req.Header.Get("Authorization")
        if !strings.HasPrefix(token, "Bearer ") {
                err = fmt.Errorf("missing token")
                return
        }
        token = token[7:]

        // Check token cache
        if tokenData := l.getToken(token); tokenData != nil {
                return tokenData.podname, tokenData.sid, tokenData.ns, nil
        }
```

At this line, Security Guard has not checked the length of the token, and it may be shorter than 7 characters. If it is, Go will panic with an out of bounds panic. This panic is recoverable and the impact is limited.

Missing SECURITY.md file

ID	ADA-KNATIVE-23-12
Component	Security Guard
Severity	Informational
Status: Fixed	

Knative offers a way to disclose security issues, but this is currently not communicated at a repository level.

Not having a security policy can result in Knative missing out on valuable community-driven security contributions and disclosures. Security researchers who wish to do their own auditing of the Knative ecosystem and who may have identified security vulnerabilities in the Knative code base will not know where or to whom to disclose their findings. Certainly, disclosing potential security-critical bugs in the Knative subprojects' public Github issues is not an approach many researchers will take. Without a security policy, it is difficult to guess who from a given Knative subproject is trusted enough and has enough bandwidth to process incoming security disclosures. This in itself has the dilemma of a responsible disclosure timeline; i.e. when security contributors may make their findings public after disclosing it to Knative; Most projects follow an industry standard of 90 day responsible disclosure timeline; however users will now know what Knatives is in case of a non-existent security policy. Furthermore, without a security policy, contributors will not know what constitutes the start of the responsible disclosure timeline: In the case of the 90 days, when do they start?

We recommend that each Knative subproject adds a security policy, whether they are similar or identical. Ideally, the security policy should be readily available for the community, and we also recommend placing the security policy at the root directory of each Knative repository. Each repository's security policy should contain a link to the part of the documentation where the community can disclose vulnerabilities (https://knative.dev/docs/reference/security/#security-working-group).

Possible DoS in Security Guard /sync endpoint

ID	ADA-KNATIVE-23-13
Component	Security Guard
Severity	Moderate
Status: Fixed	

An attacker who can send requests to Security Guards `/sync` endpoint can cause a resource exhaustion denial of service attack by sending an HTTP request containing a large body. Security Guard guard-service will read the entire body into memory, and Golang will perform a SigKill of guard-service as a result.

`guard-service` reads the request body entirely into memory on the line highlighted below:

https://github.com/knative-extensions/security-guard/blob/9dd8b30c7c1e9cd31bbb88898c8228a41919e690/cmd/guard-service/main.go#L236-L260

```go
func (l *learner) processSync(w http.ResponseWriter, req *http.Request) {
        var syncReq spec.SyncMessageReq
        var syncResp spec.SyncMessageResp

        record, podname, err := l.baseHandler(w, req)
        if err != nil {
                return
        }
        if req.Method != "POST" || req.URL.Path != "/sync" {
                http.Error(w, "404 not found.", http.StatusNotFound)
                return
        }

        if req.ContentLength == 0 || req.Body == nil {
                http.Error(w, "400 not found.", http.StatusBadRequest)
                return
        }

        err = json.NewDecoder(req.Body).Decode(&syncReq)
        if err != nil {
                pi.Log.Infof("processSync error: %v", err)
                http.Error(w, err.Error(), http.StatusBadRequest)
                return
        }
```

Possible DoS in Security Guard /mutate endpoint

ID	ADA-KNATIVE-23-14
Component	Security Guard
Severity	Moderate
Status: Fixed	

An attacker who can send requests to Security Guards `/mutate` endpoint can cause a resource exhaustion denial of service attack by sending an HTTP request containing a large body. Security Guards `guard-service` will read the entire body into memory, and Golang will perform a SigKill of guard-service as a result.

Security Guard reads the request body entirely into memory on the line highlighted below:

https://github.com/knative-extensions/security-guard/blob/9dd8b30c7c1e9cd31bbb88898c8228a41919e690/cmd/guard-webhook/main.go#L63C1-L77C1

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

        // verify the content type is accurate
        contentType := r.Header.Get("Content-Type")
        if contentType != "application/json" {
                Log.Error("contentType=%s, expect application/json", contentType)
                return
        }
```

Potential slowloris attacks in Eventing-Gitlab

ID	ADA-KNATIVE-23-15
Component	Eventing-Gitlab
Severity	Low
Status: Fixed	

Slowloris is a type of attack where an attacker opens a connection between their controlled machine and the victim's server. Once the attacker has opened the connection, they keep it open for as long as possible. They will do the same with a large number of controlled machines to hog the available connections and prevent other users from accessing the service. As such, the victim's server stays up but remains busy from processing the attacker's requests and becomes unavailable to legitimate users.

An attacker can exploit a Slowloris issue by identifying execution paths in their target application that cause it to take longer time to return from, and the attacker can then send requests that force the application into these. The fact that the Eventing-Gitlab server is susceptible to a Slowloris attack does not mean that it is easily exploitable.

The following server does not set a `ReadHeaderTimeout`, which could lead to a DDoS attack, where a large group of users send requests to the server, causing the server to hang for long enough to deny it from being available to other users, also known as a Slowloris attack:

https://github.com/knative-extensions/eventing-gitlab/blob/3221536fea4ea5b60ac06ef701d01411f9453c7d/pkg/adapter/receive_adapter.go#L94-L97

```
	server := &http.Server{
		Addr:    ":" + ra.port,
		Handler: ra.newRouter(hook),
	}
```

Hard-coded insecure protocol used by Knative Serving Activator

ID	ADA-KNATIVE-23-16
Component	Knative Serving
Severity	Low
Status: Reported	

The revisionWatcher of the Knative Serving Activator uses the HTTP protocol when probing the destination. This could allow an attacker to perform a Man-in-the-middle and return incorrect information that seemingly originates from the destination.

The root cause of the issue is that the HTTP scheme is hard-coded in the `url.Url`:

https://github.com/knative/serving/blob/d6c833f98f7abff3d183553d5f0bf01d529d4a84/pkg/activator/net/revision_backends.go#L163C1-L169C1

```
func (rw *revisionWatcher) probe(ctx context.Context, dest string) (pass bool, notMesh
bool, err error) {
        httpDest := url.URL{
                Scheme: "http",
                Host:   dest,
                Path:   nethttp.HealthCheckPath,
        }
```

The destination is always a ClusterIP Service or PodIPs, so the ability to inject traffic within the Cluster IP space implies compromise of the CNI layer.

The Knative team has triaged this issue and has not found any immediate exploitability. The Knative team will triage this further to investigate how to fix this in the optimal way.

Knative static analysis tooling

In this section, we include our observations concerning Knative's static security testing suite. We include these as a suggestion for future work on improving Knative's security posture.
Static analysis tools are useful for detecting potential security issues during the development lifecycle and in production code. When running in the CI, they can test new code contributions for security issues and help prevent these from getting merged into the codebase.

At a high level, the core Knative projects - Serving and Eventing - have a mature static toolchain that both runs in the CI and includes inline comments to disable noise for false positives, whereas most of the Knative-Extensions projects and Knative Func had no static security tooling integrated. We ran several static tools against the projects in scope and found a few true-positive security issues that we have included in the findings in this report.

As a general goal, we recommend the Knative runs the following static security tools in their CI pipeline:

1. Gosec
2. CodeQL
3. Semgrep with selected rules that test for high-impact risks and have a low level of false positives.

In addition, we recommend Knative adopts other, more cloud-oriented static tools to test for security issues in resources such as:

1. Checkov
2. KubeAudit
3. KubeScape

These tools are useful for the Knative community to reason about the Knative cluster resources. For example, a Kubescape scan[5] across all Knative code assets in scope found that several resources include possible security risks:

Severity	Control Name	Failed Resources	All Resources	% Compliance-Score
High	Resource limits	49	78	37%
High	Applications credentials in configuration files	4	219	98%
High	Host PID/IPC privileges	0	78	100%

[5] v2.9.1

High	HostNetwork access	0	78	100%
High	Insecure capabilities	0	78	100%
High	Privileged container	0	78	100%
High	CVE-2021-25742-nginx-ingress-snippet-annotation-vulnerability	0	0	100%
Medium	Exec into container	0	163	100%
Medium	Non-root containers	35	78	55%
Medium	Allow privilege escalation	34	78	56%
Medium	Ingress and Egress blocked	75	78	4%
Medium	Automatic mapping of service account	54	131	59%
Medium	Cluster-admin binding	0	163	100%
Medium	Container hostPort	0	78	100%
Medium	Cluster internal networking	23	23	0%
Medium	Linux hardening	35	78	55%
Low	Immutable container filesystem	35	78	55%
Low	PSP enabled	0	0	100%

Furthermore, integrating these tools may have implications on adoption; Kubescape tests for the security guidelines set forth by NSA and CISA in the Kubernetes Hardening Guidance. Proving that Knative adheres to these guidelines may increase adoption across critical industries required to comply with the Kubernetes Hardening Guidance from internal policies, downstream users or public regulations.