ADA LOGICS

Kubeflow Security Audit

In collaboration with CNCF, OSTIF and the Kubeflow maintainers

Adam Korczynski, David Korczynski, Ada Logics

22nd September 2025

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

About OSTIF

The Open Source Technology Improvement Fund (OSTIF) is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

Contents

About Ada Logics **1**

About OSTIF **2**

Audit contacts **5**

Introduction **6**
- Risk scoring . . . 7
- Scope . . . 7

Scorecard **8**
- Katib (6.5/10) . . . 8
- Trainer (6.1/10) . . . 9
- Spark Operator (6.9/10) . . . 10
- Notebooks (5.6/10) . . . 11
- Model Registry (7.7/10) . . . 11
- Pipelines (5.8/10) . . . 12

CI Testing **14**

Fuzzing **15**

Threat model **16**
- Pipelines . . . 17
- Katib . . . 18
- Model Registry . . . 19
- Notebooks . . . 20
- Spark Operator . . . 22
- Trainer . . . 22

Found issues **24**

Command injection can lead to RCE in Pipelines's CI **25**

Attacker can leak GitHub secret leading to repository takeover **26**

Code intends to authenticate but does not do so **27**

Server-Side Request Forgery (SSRF) in Kubeflow Pipelines CreatePipelineV1 API **30**

SSRF in Kserve HTTPSProvider **33**

Path Traversal in KServe GCS Downloader **34**

Kubeflow Pipelines: MinioObjectStore.GetFile – Reliability & Security Issues **38**

Pipelines v2 component launcher users vulnerable to path traversal from malicious Object name **39**

Resources missing network policies **42**

Model-Registry enables profiling endpoints by default **43**

Katib uses weak image reference validation **44**

Possible DoS from malicious tar archive **45**

Private key included in source code **47**

Stack-overflow bugs in 3rd-party dependency **48**

Audit contacts

Contact	Role	Organisation	Email
Adam Korczynski	Auditor	Ada Logics Ltd	adam@adalogics.com
David Korczynski	Auditor	Ada Logics Ltd	david@adalogics.com
Amir Montazery	Facilitator	OSTIF	amir@ostif.org
Derek Zimmer	Facilitator	OSTIF	derek@ostif.org
Helen Woeste	Facilitator	OSTIF	helen@ostif.org
Tom Welter	Facilitator	OSTIF	tom@ostif.org
Julius von Kohout	Kubeflow Maintainer	Kubeflow	juliusvonkohout@gmail.com
Matthew Wicks	Kubeflow Maintainer	Kubeflow	mathew.wicks@gmail.com
Andrey Velichkevich	Kubeflow Maintainer	Kubeflow	andrey.velichkevich@gmail.com

Introduction

In August and September 2025, Ada Logics carried out a holistic security audit of six projects of the Kubeflow ecosystem. The goals of the audit was to carry out threat modelling, manual code auditing and set up initial fuzzing infrastructure. The audit was facilitated by the Open Source Technology Improvement Fund (OSTIF) and funded by the Cloud Native Computing Foundation (CNCF). Kubeflow maintainers from each of the six projects also took part. Two researchers from Ada Logics worked on the audit.

The audit started with a kickoff meeting to discuss strategy. Most of the goals of the audit had been defined prior to the audit starting, and the kickoff meeting mainly formed the practical details of the audit such as communication channels and timeline.

This report describes our findings from the audit. The findings can be categorized as follows:

1. We threat modelled the six projects in scope to understand threats, attack surface, attact vectors and threat actors.
2. We have assessed the projects in scopes CI testing and made suggestions for improvements.
3. We have made pull requests with the Scorecard CI workflow to shift supply-chain security left in Kubeflows software development lifecycle.
4. We have set up fuzzing for four of the projects including OSS-Fuzz integrations and fuzz coverage.
5. We manually audited the Kubeflow source code and shared the findings with the Kubeflow team.

We found a total of 14 security issues during the audit of varying severity and type.

Risk scoring

We use a simplified risk scoring system that considers risk exposure and risk impact. Exposure is the level at which an issue is exposed to an attacker. Impact is the level of privilege escalation an attacker can obtain by exploiting the security issue. We score both on a scale of 1-5 and add the two scores together for a final combined score. This score determines the severity of security issues. We assign the severity to the issues we find.

Risk Exposure

5: The security issue exists in core component(s) and is exposed in all use cases to untrusted input. 4: The security issue exists in widely used component(s) and is enabled by default. Users of the component(s) expose the issue by default to untrusted input. 3: The issue is exposed to authenticated and/or authorized users only. 2: The issue exists in component(s) that users need to enable to be affected. 1: The issue is only exposed to trusted users.

Risk Impact

5: An attack will have the highest possible impact. 4: An attack will have high impact with some constraints or limitations. 3: An attack can cause partial harm. 2: An attack can result in privilege escalation that will cause limited harm. 1: An attack can result in limited privilege escalation but requires further privilege escalation to cause harm.

We score each issue on both scales and then add the scores for a combined total score. The total score is the basis for the overall severity of found issues.

- 10: Critical
- 9 - 8: High
- 7 - 6: Moderate
- 5 - 4: Low
- 3 - 1: Informational

Scope

The audit included the code in the following code repositories:

1. https://github.com/kubeflow/pipelines
2. https://github.com/kubeflow/model-registry
3. https://github.com/kubeflow/notebooks
4. https://github.com/kubeflow/spark-operator
5. https://github.com/kubeflow/trainer
6. https://github.com/kubeflow/katib

Scorecard

During the audit we assessed the six projects in scope by way of OpenSSF Scorecard.

OpenSSF Scorecard is a tool that automatically evaluates open-source projects for supply-chain security best practices. It highlights where projects are strong, where they fall short, and how they can improve, making the broader software ecosystem more resilient against supply chain and application-level attacks. By scanning a project's repository, it generates a set of scores (0-10) across multiple categories, offering developers, users, and organizations a quick way to assess how trustworthy and secure a project appears.

Each check maps to common weak points attackers have exploited in real-world incidents. Some examples of the Scorecard checks are:

1. **Branch Protection**: Ensures important branches like main cannot be directly overwritten. Weak protections can allow unnoticed malicious changes.
2. **Code Review**: Confirms pull requests are reviewed before merging. Oversight makes it harder for malicious code to slip in. In the event-stream npm compromise (2018), a malicious maintainer added a malicious dependency (flatmap-stream) without the scrutiny of a review process which made the attack possible.
3. **Dependency Management**: Ensures projects keep dependencies updated. A known attack of this type is the Equifax breach (2017) stemmed from an unpatched Apache Struts flaw (CVE-2017-5638) that attackers exploited months after a fix was available (BlackDuck).
4. **Continuous Integration (CI)**: Verifies automated builds and tests are in place, reducing the chance insecure or tampered code makes it to release.

Across all six projects, strong practices in code review, contributors, and CI testing provide a solid foundation. But every project struggles with fuzzing, nearly all fail to pin dependencies, and all six projects fail to define high-level or job-level token permissions which could result in overprivileging tokens in CI workflows. Where vulnerability scores are low, it is generally due to outdated or insecure dependencies. Below, we enumerate the Scorecard results of each of the six projects. Note that we made pull requests to each project to add the Scorecard workflow:

- Pipelines: https://github.com/kubeflow/pipelines/pull/12229
- Model Registry: https://github.com/kubeflow/model-registry/pull/1580
- Notebooks: https://github.com/kubeflow/notebooks/pull/580
- Spark Operator: https://github.com/kubeflow/spark-operator/pull/2654
- Trainer: https://github.com/kubeflow/trainer/pull/2824
- Katib: https://github.com/kubeflow/katib/pull/2569

Katib (6.5/10)

SCORE	NAME	RISK LEVEL
10 / 10	Binary-Artifacts	High
?	Branch-Protection	High
10 / 10	CI-Tests	Low
10 / 10	CII-Best-Practices	Low
10 / 10	Code-Review	High
10 / 10	Contributors	Low
10 / 10	Dangerous-Workflow	Critical

10 / 10	Dependency-Update-Tool	High
0 / 10	Fuzzing	Medium
10 / 10	License	Low
10 / 10	Maintained	High
?	Packaging	Medium
0 / 10	Pinned-Dependencies	Medium
1 / 10	SAST	Medium
10 / 10	Security-Policy	Medium
?	Signed-Releases	High
0 / 10	Token-Permissions	High
0 / 10	Vulnerabilities	High

Katib demonstrated strong practices in the code review, CI testing, contributor activity, and security policy checks, where it achieved perfect scores. However, it struggles with fuzzing, pinned dependencies, token permissions, and vulnerability management. The pinned dependencies check revealed 90 unpinned references across 38 files, with most issues concentrated in Dockerfiles and GitHub workflows. This means Katib often relies on floating versions of dependencies, leaving it exposed if an upstream action or base image changes unexpectedly. With regards to token permissions: 17 GitHub workflows lacked top-level permission definitions, which means the default GitHub token has more privileges than may be necessary. This increases the risk of a compromised workflow being able to push malicious changes. Finally, Katib's vulnerability score is low because its dependency files included 34 packages with known vulnerabilities.

Trainer (6.1/10)

SCORE	NAME	RISK LEVEL
10 / 10	Binary-Artifacts	High
?	Branch-Protection	High
10 / 10	CI-Tests	Low
10 / 10	CII-Best-Practices	Low
10 / 10	Code-Review	High
10 / 10	Contributors	Low
0 / 10	Dangerous-Workflow	Critical
10 / 10	Dependency-Update-Tool	High
0 / 10	Fuzzing	Medium
10 / 10	License	Low
10 / 10	Maintained	High
?	Packaging	Medium
0 / 10	Pinned-Dependencies	Medium
0 / 10	SAST	Medium
10 / 10	Security-Policy	Medium
?	Signed-Releases	High

0 / 10	Token-Permissions	High
9 / 10	Vulnerabilities	High

Trainer scored well in reviews, contributors, and dependency update tooling but is weakened by insecure workflow design, missing fuzzing, and dependency hygiene issues. Pinned dependencies were absent across multiple workflows and Dockerfiles, showing the same reliance on floating versions as Katib. Several workflows were also missing top-level permissions, while one workflow explicitly set `"actions"` permissions to write, unnecessarily granting higher privileges. These two issues combined mean that a compromised workflow could have more impact than it needs. Despite these weaknesses, Trainer's vulnerability score is strong (9/10), suggesting its dependencies were mostly free of known CVEs at the time of scanning.

Spark Operator (6.9/10)

SCORE	NAME	RISK LEVEL
10 / 10	Binary-Artifacts	High
?	Branch-Protection	High
10 / 10	CI-Tests	Low
5 / 10	CII-Best-Practices	Low
10 / 10	Code-Review	High
10 / 10	Contributors	Low
10 / 10	Dangerous-Workflow	Critical
10 / 10	Dependency-Update-Tool	High
0 / 10	Fuzzing	Medium
10 / 10	License	Low
10 / 10	Maintained	High
10 / 10	Packaging	Medium
0 / 10	Pinned-Dependencies	Medium
3 / 10	SAST	Medium
10 / 10	Security-Policy	Medium
0 / 10	Signed-Releases	High
0 / 10	Token-Permissions	High
10 / 10	Vulnerabilities	High

Spark Operator was one of the stronger projects overall with regards to its Scorecard results, performing well in reviews, CI tests, contributor activity, and maintenance. It also uses dependency update tooling effectively. Its testing suite was missing fuzzing, and scored low in other checks due to unsigned releases, and widespread unpinned dependencies. The pinned dependencies check flagged unpinned references in major workflows like releases, integrations, and Dockerfiles, showing a recurring pattern of relying on floating versions. Token permissions are also missing in several workflows, meaning they default to overly broad GitHub token access. These gaps could allow an attacker to exploit workflows or dependency drift, even though Spark Operator's vulnerability score remains high.

Notebooks (5.6/10)

SCORE	NAME	RISK LEVEL
10 / 10	Binary-Artifacts	High
?	Branch-Protection	High
6 / 10	CI-Tests	Low
2 / 10	CII-Best-Practices	Low
8 / 10	Code-Review	High
0 / 10	Contributors	Low
10 / 10	Dangerous-Workflow	Critical
0 / 10	Dependency-Update-Tool	High
0 / 10	Fuzzing	Medium
10 / 10	License	Low
10 / 10	Maintained	High
?	Packaging	Medium
0 / 10	Pinned-Dependencies	Medium
0 / 10	SAST	Medium
10 / 10	Security-Policy	Medium
?	Signed-Releases	High
0 / 10	Token-Permissions	High
10 / 10	Vulnerabilities	High

Notebooks earned good marks for licensing, maintenance, and security policy, but struggled in almost every other area. Its contributor score is zero, automated dependency management is absent, and it lacks fuzzing, SAST, and update tooling. The pinned dependencies. Token permissions are also weak: one workflow explicitly set actions to write, while others define no permissions at all. These gaps give workflows unnecessary privileges and increase the risk of a supply chain compromise. Interestingly, the project's vulnerability score is perfect (10/10).

Model Registry (7.7/10)

SCORE	NAME	RISK LEVEL
10 / 10	Binary-Artifacts	High
?	Branch-Protection	High
10 / 10	CI-Tests	Low
5 / 10	CII-Best-Practices	Low
10 / 10	Code-Review	High
10 / 10	Contributors	Low
10 / 10	Dangerous-Workflow	Critical

10 / 10	Dependency-Update-Tool	High
0 / 10	Fuzzing	Medium
10 / 10	License	Low
10 / 10	Maintained	High
10 / 10	Packaging	Medium
0 / 10	Pinned-Dependencies	Medium
4 / 10	SAST	Medium
10 / 10	Security-Policy	Medium
?	Signed-Releases	High
5 / 10	Token-Permissions	High
6 / 10	Vulnerabilities	High

Model Registry was the highest scoring project at 7.7. It scored 10/10 in code reviews, contributors, dependency tooling. Its CI and policy enforcement practices were also strong which signals good project health. It was lacking in fuzzing, static analysis, and vulnerability management. The pinned dependencies check revealed a high number of unpinned references across workflows and Dockerfiles, which could allow unexpected changes from upstream dependencies. Several workflows also lacked top-level token permissions, again leaving the default GitHub token with more privileges than necessary. Its vulnerability score of 6 suggests that while many practices are strong, dependency files include some packages with known CVEs that need addressing.

Pipelines (5.8/10)

SCORE	NAME	RISK LEVEL
10 / 10	Binary-Artifacts	High
?	Branch-Protection	High
10 / 10	CI-Tests	Low
5 / 10	CII-Best-Practices	Low
9 / 10	Code-Review	High
10 / 10	Contributors	Low
0 / 10	Dangerous-Workflow	Critical
10 / 10	Dependency-Update-Tool	High
0 / 10	Fuzzing	Medium
10 / 10	License	Low
10 / 10	Maintained	High
?	Packaging	Medium
? / 10	Pinned-Dependencies	Medium
7 / 10	SAST	Medium
10 / 10	Security-Policy	Medium
?	Signed-Releases	High
0 / 10	Token-Permissions	High

0 / 10	Vulnerabilities	High

Pipelines showed strong engagement in code review, CI testing, and contributor activity, but its security posture was lacking from missing fuzzing, a failed dangerous workflows check, and poor dependency hygiene. The pinned dependencies result was inconclusive, but token permissions showed that over thirty GitHub workflows lacked explicit permission settings, which is the most widespread issue among all projects reviewed. This default over-privilege could give an attacker broad access if they were able to compromise a workflow. The project also scored zero in the Vulnerabilities check because its dependency files contained packages with known CVEs. Combined with unpinned or unmanaged dependencies, this creates a risk of supply-chain attacks through Pipelines's dependency tree that can easily be mitigated.

CI Testing

A minor part of our time was spent on reviewing Kubeflows CI testing. The motivation behind this was to assess if Kubeflow could make any obvious, low to medium effort improvements that would have a significant improvement on Kubeflows security. All Kubeflow projects have extensive CI test suites, and as such, potential improvements are to further enhance the CI. We recommend adding the following tools to each Kubeflow sub project:

1. **Gosec**: The projects in scope are implemented in Go, and as such they can all benefit from continuous testing by one of the most popular security SAST tools for Go-based applications: Gosec. One of the findings in this audit - ADA-KUBEFL-12 - was found with the help of Gosec. We recommend adding the Gosec GitHub workflow to the Kubeflow projects' CI to catch coding mistakes that weaken Kubeflows security posture. The effort will not be trivial, as Gosec does not consider Kubeflows threat model and as a result reports false positives, however, it also offers suggestions that will harden Kubeflows security once implemented.
2. **Semgrep**: Semgrep is a tool similar to Gosec in that it is a SAST tool for code-level bugs. Semgrep and Gosec have different rulesets and can be used alongside each other with benefit.
3. **Scorecard**: Scorecard is a tool for assessing supply-chain related heuristics of a software package. It analyzes heuristics in the software development life cycle, the CI, dependencies and more. It can be used as a command-line tool and also offers a CI workflow to shift supply-chain security analysis left. By adding the Scorecard workflow, Kubeflow can identify unhardened supply-chain areas present right now, and with the workflow, it can avoid reducing mitigation and hardening over time. We added the Scorecard workflow to the projects in scope of this audit with the following pull requests:

    1. https://github.com/kubeflow/pipelines/pull/12229
    2. https://github.com/kubeflow/model-registry/pull/1580
    3. https://github.com/kubeflow/notebooks/pull/580
    4. https://github.com/kubeflow/spark-operator/pull/2654
    5. https://github.com/kubeflow/trainer/pull/2824
    6. https://github.com/kubeflow/katib/pull/2569

Fuzzing

During the security audit, we added fuzz testing to four of the six projects in scope. We found that fuzzing would not be beneficial to the two projects which we left out. We leveraged coverage guided fuzz testing and OSS-Fuzz to create a sustainable infrastructure for each of the four projects, such that they maintain their productivity after the audit has ended.

We integrated the following four projects into OSS-Fuzz:

1. Katib. Integration PR: https://github.com/google/oss-fuzz/pull/13962
2. Pipelines. Integration PR: https://github.com/google/oss-fuzz/pull/13963
3. Spark Operator. Integration PR: https://github.com/google/oss-fuzz/pull/13960
4. Model Registry. Integration PR: https://github.com/google/oss-fuzz/pull/13961

Along with the infrastructure to support the OSS-Fuzz integration, we added fuzz coverage for each project. We primarily prioritized coverage of complex processing routines which is where coverage guided fuzz testing excels at, however, the projects can expand to other ways of using fuzzing by testing the controllers of each project.

The integration into OSS-Fuzz will help with the four projects' Scorecard score; all six projects scored low in the fuzzing check, and the OSS-Fuzz integration improves on that.

Model Registrys fuzzer found three crashes in 3rd-party dependencies. The three crashes are all non-recoverable meaning that they would be able to cause denial of service of the affected component, however, the input in this particular case is trusted and as such it merely represents a functional bug. In addition, the crashes also serve as a reminder that it is a good idea to cover 3rd party dependencies that carry out complex processing with fuzzing. We recommend following such principle moving forward.

Threat model

In this section we enumerate the projects and discuss their threat models. Threat modelling is a high-level exercise that describes the types of threats the Kubeflow projects face, which threat actors can weaponize these threats and where and how these threat actors would do so in Kubeflow. Threat modelling is not about discussing concrete, existing vulnerabilities. Instead, it helps us to reason about how we should think about the projects' security, what we should look for at a high level and where we should look.

In each project's threat model we discuss its threats and the impact the exploitation can have if a threat actor would be able to exploit vulnerabilities in the project. Each project's threat model focuses on runtime threats, however all projects face threats in their supply-chain and in areas such as misconfiguration. We have discussed the supply-chain threats in the Scorecard section above.

Before we discuss the details of each of the Kubeflow projects in scope, we will first enumerate the threat actors in Kubeflows threat model. A threat actor is first and foremost any actor that can affect the security of Kubeflows security. This does not mean or imply they will, but simply that they can. It is important to consider all threat actors from the perspective of how they can impact Kubeflows security and how Kubeflow should consider a threat from each.

We summarize the threat actors that can impact Kubeflows security in the table below sorted by privilege.

#	Name	Level of trust	Type of threat
1	Cluster admin	High	Runtime
2	Kubeflow maintainer	High	Supply-chain
3	Maintainer of external services	Low	Runtime
4	Kubeflow contributor	Low	Supply-chain
5	Contributor to external services	Low	Runtime
6	Cluster user	Low	Runtime
7	Maintainer of 3rd-party dependencies	None	Supply-chain
8	Contributors to 3rd-party dependencies	None	Supply-chain

#1 Cluster Administrator The cluster administrator holds the highest level of privilege within a Kubeflow deployment. This role includes the ability to configure, start, and shut down the cluster. Because administrators already operate with full control, attacks requiring cluster-admin privileges are generally of limited concern for Kubeflow's threat model - there is little opportunity for privilege escalation beyond the cluster admins existing privileges.

#2 Kubeflow Maintainer Kubeflow maintainers can influence users indirectly by modifying the source code that downstream deployments consume. While they cannot typically alter user deployments directly, they occupy a position of elevated trust.

Maintainers may act maliciously - motivated by financial, political, or personal incentives - or may become victims of social engineering, phishing, or impersonation. Overprivileged maintainer permissions significantly increase this risk. Dangerous examples include:

1. Direct push access to the main branch. This could be used to push malicious code in an unnoticed manner, skipping CI testing and reviews.
2. The ability to merge their own pull requests without peer review.
3. Authority to modify official releases without oversight.

#3 Maintainer of External Services and #5 Contributor to External Services Kubeflow projects often integrate with external services such as cloud storage. Maintainers or contributors to these services may not belong to the same trust boundary as Kubeflow cluster users. Misconfigurations or insecure integrations in external services could allow these actors to escalate their privileges within a cluster.

For example, if Kubeflow processes cloud objects insecurely, a malicious actor with cloud storage access could craft objects that trigger vulnerabilities when Kubeflow processes them. Although these actors have no direct privileges in the cluster, their ability to influence inputs makes them a credible threat.

#4 Kubeflow Contributor Contributors are any GitHub users who submit code to Kubeflow repositories. Since contributions can come from untrusted individuals, this actor class poses inherent supply-chain risk.

The XZ Utils backdoor attempt (2024) illustrates the risk of untrusted contributors successfully inserting harmful code.

Contributors may also target weaknesses in source code management or continuous integration (CI) pipelines to escalate their impact.

#6 Cluster User Cluster users interact with Kubeflow workloads by creating or managing experiments, pipelines, artifacts, and other resources. Since they typically hold the lowest privileges within a deployment, they represent the higest possible privilege escalation.

While most cluster users are legitimate, they can also turn against the cluster, their employer or other users in the cluster and should not be trusted. Motivations to turn malicious may include retaliation by disgruntled employees, financial incentives from third parties or ideological reasons.

#7 Maintainer of Third-Party Dependencies and #8 Contributor to Third-Party Dependencies Maintainers and contributors to third-party dependencies pose significant supply-chain risk. By introducing vulnerabilities - intentionally or accidentally - into packages that Kubeflow consumes, they can directly affect Kubeflow's security.

The Equifax breach (2017), caused by failure to patch a vulnerability in Apache Struts, and the dependency confusion attacks (2020), where attackers exploited package naming conventions to poison dependencies, both show the critical importance of dependency hygiene [Equifax, 2017; Dependency Confusion, 2020].

Strong dependency management practices - including version pinning, vulnerability scanning, and regular audits - are essential to mitigate these risks.

Pipelines

Kubeflow Pipelines (KFP) orchestrates machine learning workflows on Kubernetes. It enables reproducibility, portability, and scaling of experiments and model training by structuring workflows into containerized pipelines.

Key Threats

1. **Weak Authentication & Authorization** - Improper enforcement of access controls - such as AuthServer bugs - could allow one user to view or modify another user's resources.
2. **Data Exposure & Insider Threats** - Unauthorized access to sensitive experiments or artifacts may lead to regulatory breaches, reputational harm, or insider misuse.
3. **Resource Tampering & Disruption** - Malicious users could delete, corrupt, or alter others' pipelines, resulting in misleading outputs or operational downtime.
4. **Frontend Vulnerabilities** - Poorly validated resource data may enable cross-site scripting (XSS) or UI-driven privilege escalation.
5. **API Server Exploits** - Vulnerabilities in request handling could expose privilege escalation paths to all users.
6. **External Services & Object Storage** - Misconfigured integrations or unvalidated objects from cloud storage could enable path traversal or remote code execution, even without cluster-level permissions.

Potential Impacts:

- Unauthorized disclosure of sensitive data.
- Corruption or loss of experiments and artifacts.
- Misleading or sabotaged ML model outputs.
- Escalation from unprivileged to administrator access.
- Regulatory and contractual violations leading to financial or reputational damage.

Because KFP is often deployed in a multi-user environment, it must enforce strict security boundaries between users. Multiple individuals interact with the same backend and frontend, and without proper authorization controls, one user could interfere with another's resources.

At the center of this boundary is the AuthServer, a critical component responsible for verifying whether a user has permission to perform a specific action on a resource. For instance, if a user attempts to modify or delete a pipeline run, the AuthServer checks their privileges and either allows or denies the operation.

The AuthServer is essential for maintaining isolation between users, but it is not the only defense. Other components that process user requests must also be hardened. A single bug in request handling could allow a malicious user to bypass the AuthServer's checks and gain access to resources they should not control. For this reason, Pipelines security relies on both a strong AuthServer and robust security practices throughout the codebase.

Attack Surfaces

1. Kubeflow API Server The Pipelines backend exposes APIs that process untrusted user requests. Execution paths that process cluster objects must be secure against privilege escalation and data tampering. Because nearly all users interact with the API, any vulnerability here could be exploited widely across the deployment.
2. External Services and Object Storage Kubeflow Pipelines integrates with cloud-based object storage services to manage datasets, models, and artifacts. This integration introduces another attack surface:

- Mishandling of filenames or metadata could result in path traversal vulnerabilities, potentially leading to arbitrary code execution.
- Malicious objects placed in cloud storage could exploit weaknesses when retrieved by Pipelines.

An important detail is that attackers leveraging cloud storage do not need cluster-level permissions. From the cluster's point of view, such attacks originate from outside its trust boundary. If Pipelines performs insufficient validation of incoming objects, attackers who control cloud storage services could escalate privileges dramatically—from having no access at all to potentially impersonating administrators. Cloud storage services do have their own authentication and authorization systems, but these are entirely separate from Kubeflow's. Misconfigurations in cloud storage can therefore translate directly into security failures inside Kubeflow.

Katib

Katib provides AutoML capabilities within Kubeflow, allowing users to run hyperparameter tuning, neural architecture search, and related optimization tasks at scale.

Key Threats 1. **Weak Authentication** - Katib implements authentication and authorization which are security features and should be scrutinized. 2. **Unauthorized Access** - Attackers could view, modify, or delete other users' experiments and results. 3. **Experiment Pods** - Malicious or over-privileged jobs could escalate privileges or misuse resources. 4. **Frontend Risks** - Insufficient input sanitization could enable XSS or client-side injection.

Potential Impacts 1. Execution of arbitrary code within the cluster. 2. Exfiltration of sensitive data or proprietary models. 3. Sabotage or disruption of research and experiments. 4. Compromise of downstream ML lifecycle through poisoned models or artifacts.

Attack Surfaces

Katib's primary attack surfaces include:

1. **Kubernetes API** - Processes objects from the cluster. Vulnerabilities here could allow privilege escalation or bypass of authentication.
2. **Experiment Pods** - Execute user-submitted code. Weak sandboxing or over-privileged configurations could lead to lateral movement within the cluster.
3. **Katib Frontend** - Displays experiment data. If input is not sanitized, attackers may execute client-side attacks against other users.

One of the ways users interact with Katib is through the Kubernetes API. Users specify details such as the parameters to optimize, the search algorithms to apply, and the metrics used for evaluation. Katib accepts these definitions and turns them into jobs that are scheduled on Kubernetes.

Authorization is crucial to Katibs security model. If it fails, an attacker could view the results of experiments belonging to other users. These results may include sensitive datasets, proprietary model parameters, or intellectual property. In many industries, such leaks could also create regulatory or contractual violations - for example, if employees without proper clearance gain access to sensitive information.

In addition, an attacker could modify or delete experiments belonging to others. This could be used to sabotage projects, corrupt results, or disrupt the productivity of entire teams. For instance, imagine an attacker who modifies hyperparameter configurations so that training jobs consistently produce suboptimal models. The results may look valid at first glance, but the underlying sabotage could delay product releases or degrade performance in production systems.

Katib also provides a frontend UI for submitting experiments and viewing results. If it does not properly sanitize input, they may be vulnerable to injection attacks such as cross-site scripting (XSS). For example, if a malicious user names an experiment with HTML or JavaScript payloads and that name is rendered unsafely in the UI, the payload could execute in the browser of another user.

This type of vulnerability can lead to session hijacking, impersonation, or privilege escalation. In multi-user environments, where many individuals share the same frontend, the consequences can spread quickly, allowing one malicious user to affect many others.

Model Registry

Kubeflow Model Registry manages the lifecycle of machine learning (ML) models, acting as a central repository for storage, versioning, and sharing across teams. It is tightly integrated into the ML pipeline: models trained through experiments in Katib or Pipelines are registered here, and production-serving systems may later retrieve models directly from it. Because of this central role, the registry is both highly valuable and highly sensitive. If compromised, it can become a distribution channel for tampered models, spreading malicious artifacts across the wider Kubeflow ecosystem and even into production environments.

Key Threats

1. **Unauthorized Access** - Weak access controls could allow theft or deletion of valuable models.
2. **Model Tampering** - Attackers could replace or alter models, inserting backdoors or sabotaging performance.
3. **Metadata Exposure/Manipulation** - Sensitive provenance data may leak, or falsified metadata could cause malicious models to appear legitimate.
4. **Integration Risks** - Models could be configured to exploit security vulnerabilities when other users download them.

Potential Impacts

- Theft of intellectual property and sensitive models.

- Deployment of backdoored or manipulated models into production.
- Loss of trust in experiment provenance and lifecycle integrity.
- Regulatory or contractual breaches if sensitive training data is exposed.
- Widespread downstream compromise through poisoned artifacts.

At the core of the Model Registry is its ability to store and retrieve models. In multi-user environments, strict access control is required to ensure that users can only view, modify, or delete models they are authorized to manage. If access controls are too permissive, one user could overwrite another's model, delete critical assets, or exfiltrate proprietary information.

This risk is amplified by the fact that models often embody sensitive intellectual property. For many organizations, trained ML models are the most valuable output of their research and development. Unauthorized access could result in the theft of trade secrets or even regulatory violations if models trained on sensitive datasets are exposed to unauthorized individuals.

The registry's primary function is to store serialized ML models. These models often represent significant intellectual property - weeks or months of training on expensive infrastructure, using proprietary datasets. If attackers gain the ability to tamper with models inside the registry, they can cause serious harm.

An attacker could subtly alter weights so that the model performs correctly under normal conditions but misclassifies specific inputs in ways beneficial to the attacker. Alternatively, the model could be modified to execute malicious code when deserialized, exploiting vulnerabilities in libraries that process the model format. In either case, trust in the ML workflow would be undermined.

Many Kubeflow deployments configure the Model Registry to store model artifacts in cloud object storage. This creates another potential attack surface: if attackers gain control of the cloud storage bucket - or exploit a misconfiguration - they could replace legitimate models with malicious versions.

Since the registry itself may trust the cloud storage as its backend, tampering at this layer could propagate unnoticed. An attacker would not need direct cluster access; control of the external storage alone would be sufficient to poison models and potentially compromise workloads once those models are retrieved.

The Model Registry also provides a fronted UI for browsing and managing models. If it renders model metadata without proper validation, it could be exploited for frontend-specific attacks, such as cross-site scripting (XSS).

For example, a maliciously crafted model could include metadata fields containing embedded scripts. When displayed in the UI, these scripts could execute in the victim's browser, leading to data theft, session hijacking, or impersonation. In shared multi-user environments, such attacks could spread laterally, allowing one malicious user to compromise others.

This kind of attack does not require tampering with the model's core functionality; manipulation of its metadata alone could be enough to launch client-side exploits.

Attack Surfaces summarized 1. **Tampered Models** - Malicious modifications to stored models may cause the registry or other Kubeflow components to process them incorrectly, leading to breaches. 2. **Cloud Storage Control** - Attackers who gain control of or exploit misconfigurations in the cloud store can replace legitimate models with malicious ones. 3. **Frontend/UI Attacks** - Maliciously crafted models or metadata could exploit frontend vulnerabilities, such as XSS, to escalate privileges or impersonate users.

Notebooks

Kubeflow Notebooks provides Kubernetes-native interactive development environments (e.g., Jupyter, RStudio, VS Code). It enables users to prototype, analyze data, and build models before transitioning work into Pipelines, Katib, or the Model Registry.

Key Threats 1. **Kubernetes API Risks** - Bugs in Notebook controllers could be exploited for denial-of-service, information disclosure, or to run workloads under another user's identity. 2. **Frontend/Backend Vulnerabilities** - Shared CRUD backends

and services (Jupyter, Tensorboards, Volumes) are central targets. Exploits here could compromise multiple users across the cluster.

Potential Impacts

- **Service Disruption**: Attackers could trigger crashes to deny access to notebooks.
- **Data Exposure**: Misconfigured or buggy controllers could leak other users' workloads or datasets.
- **Privilege Misuse**: Malicious users could run experiments on behalf of others, violating security isolation.
- **Cluster-Wide Compromise**: Vulnerabilities in shared CRUD services could give attackers leverage over multiple users.

Developing machine learning models typically begins with interactive exploration: analyzing datasets, prototyping code, and testing early ideas. Teams need environments that allow them to experiment, visualize results, and iterate quickly before moving work into larger workflows. Running these environments manually on local machines creates challenges around reproducibility, scalability, and collaboration.

Kubeflow Notebooks addresses these challenges by providing a Kubernetes-native service for launching interactive development environments such as Jupyter, RStudio, or Visual Studio Code. Each notebook server runs inside a container, ensuring that dependencies are consistent and making it easier to share environments across teams. Users access their notebook servers through a web interface, directly integrated with the Kubernetes cluster.

Within the Kubeflow ecosystem, Notebooks serve as the interactive workspace in the machine learning lifecycle. They allow users to explore datasets, test models, and prepare code for integration into automated workflows. Work done in notebooks can naturally be transitioned to other Kubeflow components—for example, turning experiments into Pipelines for orchestration, submitting tuning jobs to Katib, or registering trained models in the Model Registry.

Attack Surfaces

Kubeflow Notebooks exposes two primary attack surfaces:

1. Kubernetes API Interaction

Notebook controllers communicate with the Kubernetes API to create, modify, and view workloads. These controllers must correctly enforce permissions and handle user requests.

- **Reliability risks**: Bugs in workload handling could cause controllers to crash. If attackers can repeatedly trigger such bugs, they could perform a denial-of-service attack, making notebooks unavailable for other users.
- **Information disclosure risks**: Incorrect request handling may cause controllers to fetch or expose data belonging to other users.
- **Privilege separation risks**: Controllers must ensure workloads are generated only on behalf of the requesting user. If this fails, one user could inadvertently or maliciously run experiments under another user's identity, breaking security isolation.

2. Frontend and Backend Services

Kubeflow Notebooks includes a set of web applications (CRUD backends) used by multiple notebook services, such as:

- Jupyter
- Tensorboards
- Volumes

All of these rely on a common CRUD backend. Because these services are central and shared among many users, they represent high-value targets for attackers. A vulnerability here could allow a malicious actor to compromise not just their own environment but also affect other users in the cluster.

Spark Operator

The Spark Operator orchestrates Apache Spark applications on Kubernetes, simplifying distributed data processing in Kubeflow.

Key Threats 1. **Controller Bugs** - Exploitable logic flaws could cause DoS, information leaks, or privilege escalation. 2. **Malicious Job Specs** - Poor validation could allow attackers to run over-privileged pods or exfiltrate data.

Potential Impacts

- Denial of service against Spark workloads.
- Unauthorized access to cluster resources.
- Data exfiltration or privilege escalation via malicious job configurations.

The Spark Operator allows users to run Apache Spark applications on Kubernetes by translating Spark job specifications into Kubernetes workloads. It simplifies deployment and scaling of distributed data processing jobs, making Spark a first-class citizen in Kubeflow environments.

Because the operator accepts user-defined job specifications and interacts directly with the Kubernetes API, it extends the cluster's attack surface. A compromised or misconfigured operator could allow attackers to escalate privileges or disrupt workloads.

Attack Surfaces

1. Kubernetes API Interaction The operator creates and manages Kubernetes resources on behalf of users. If its controllers mishandle input, attackers could:

- Trigger denial-of-service by submitting workloads that crash controllers.
- Exploit logic errors to modify or view resources belonging to other users.
- Abuse controller privileges to escalate access within the cluster.

2. Spark Jobs User-submitted Spark job workloads may include arbitrary input. If validation is weak, attackers could craft malicious job specs to:

- Launch pods with excessive permissions.
- Exfiltrate data by connecting jobs to unauthorized external endpoints.
- Mount resources or service accounts beyond their intended scope.

Trainer

Kubeflow Trainer provides Kubernetes-native orchestration of distributed ML training jobs, supporting frameworks such as PyTorch, TensorFlow, JAX, and XGBoost.

Key Threats 1. **Unsafe Controller Logic** - Cluster objects are untrusted inputs; poor sanitization may enable path traversal, incorrect workload creation, or other privilege escalations. 2. **Cluster User Abuse** - Malicious users with legitimate access could exploit logical bugs to escalate privileges, interfere with others' workloads, or cause denial-of-service.

Potential Impacts

- Privilege escalation and unauthorized access to cluster resources.
- Tampering with or disruption of other users' training jobs.
- Denial-of-service against the Trainer operator, affecting cluster reliability.
- Corruption or manipulation of training results, leading to downstream risks in Pipelines or Model Registry.

Training modern machine learning models at scale often requires distributing workloads across multiple nodes, coordinating specialized hardware such as GPUs or TPUs, and integrating advanced optimization libraries. Large language models (LLMs) and other state-of-the-art workloads demand orchestration systems that can reliably manage distributed execution while supporting frameworks designed for high-performance training.

Within the Kubeflow ecosystem, Trainer is the core distributed training component. It connects with Pipelines for orchestration, integrates with Katib for hyperparameter optimization across distributed jobs, and supports downstream model registration in the Model Registry. By treating training as a first-class Kubernetes workload, Trainer helps teams standardize training processes, fine-tune large models, and scale experiments from prototype to production.

Attack Surface and Threat Model The primary attack surface of Kubeflow Trainer lies in its interaction with the Kubernetes API. Users create and modify these workloads to define training jobs, and the Trainer's controllers are responsible for processing them. The Kubeflow Trainer controllers reside in the Kubeflow Trainer repository, where they reconcile custom resources with the desired cluster state.

Because some objects originate from user input, they must be treated as untrusted data. If the Trainer processes them unsafely, attackers could leverage the system to cause security breaches. For example:

- If fields from an object are written to the filesystem without sanitization, attackers could embed path traversal characters to overwrite files outside intended directories.
- If values are trusted without validation, attackers could craft objects that cause the controller to create workloads in unintended ways.

In short, the controller logic must sanitize and validate any user-provided data before using it.

The primary threat actor for Trainer is the cluster user. Cluster users already have legitimate access to submit workloads, but they may attempt to exploit logical bugs in order to escalate privileges or interfere with others. Some examples include:

- **Privilege Escalation**: A user crafts CRDs that trick Trainer into scheduling workloads with higher permissions than they should have, giving the attacker access to sensitive data or resources.
- **Cross-Tenant Access**: Improper permission segmentation may allow one user to read or interfere with another user's workloads.
- **Denial-of-Service**: A malicious user repeatedly submits workloads that exploit controller bugs, causing Trainer to crash or become unresponsive, thereby denying service to others.
- **Tampering with Workloads**: Attackers could modify workloads belonging to other users, altering training jobs or corrupting results.

Because of this, runtime security issues in Trainer will most often involve the cluster user as the key threat actor. While external adversaries may also pose risks if authentication or access boundaries are weak, the most realistic and immediate concern comes from malicious or compromised insiders who already operate within the cluster.

Found issues

ID	Name	Severity	Status
ADA-KUBEFL-09	Command injection can lead to RCE in Pipelines's CI	Critical	Reported
ADA-KUBEFL-10	Attacker can leak GitHub secret leading to repository takeover	Critical	Reported
ADA-KUBEFL-13	Code intends to authenticate but does not do so	Critical	Reported
ADA-KUBEFL-01	Server-Side Request Forgery (SSRF) in Kubeflow Pipelines CreatePipelineV1 API	Moderate	Reported
ADA-KUBEFL-02	SSRF in Kserve HTTPSProvider	Moderate	Reported
ADA-KUBEFL-03	Path Traversal in KServe GCS Downloader	Moderate	Reported
ADA-KUBEFL-04	Kubeflow Pipelines: MinioObjectStore.GetFile – Reliability and Security Issues	Moderate	Reported
ADA-KUBEFL-07	Pipelines v2 component launcher users vulnerable to path traversal from malicious Object name	Moderate	Reported
ADA-KUBEFL-08	Resources missing network policies	Moderate	Reported
ADA-KUBEFL-12	Model-Registry enables profiling endpoints by default	Moderate	Reported
ADA-KUBEFL-05	Katib uses weak image reference validation	Low	Reported
ADA-KUBEFL-06	Possible DoS from malicious tar archive	Low	Reported
ADA-KUBEFL-11	Private key included in source code by default	Informational	Reported
ADA-KUBEFL-14	Stack-overflow bugs in 3rd-party dependency	Informational	Reported

Command injection can lead to RCE in Pipelines's CI

Severity	Critical
Status	Reported
ID	ADA-KUBEFL-09
Component	Pipelines CI

Pipelines's CI has a script injection vulnerability that can allow users to run arbitrary commands in its CI.

https://github.com/kubeflow/pipelines/blob/99ea0c4cce235d5aaae15319541c888455d1aab0/.github/workflows/presubmit-backend.yml#L31

```
31      - name: Run Backend Tests
32        run: |
33          export GIT_BRANCH=${{ github.head_ref || github.ref_name }}
34          export GIT_REPO=${{ github.event.pull_request.head.repo.full_name }}
35          ./test/presubmit-backend-test.sh
```

The workflow interpolates untrusted GitHub context directly into a shell command inside a `run:` step. GitHub evaluates `${{ ...}}` first, and then bash parses the resulting line. In bash, the right-hand side of `NAME=value` still performs command substitution (e.g., `$(...)` or backticks) and other expansions. Therefore, if the branch/ref name contains command-substitution syntax, bash will execute it when running export.

The workflow should sanitize untrusted user input before intepreting it in the bash script. To increase security, the workflow could also impose a whitelist of allowed characters.

Attacker can leak GitHub secret leading to repository takeover

Severity	Critical
Status	Reported
ID	ADA-KUBEFL-10
Component	Trainer CI

Trainers CI is vulnerable to a GitHub token leak which can potentially lead to repository takeover.

Trainers runs scripts from untrusted repositories in the following workflow:https://github.com/kubeflow/trainer/blob/d13a831c0072fe10c7e3019aa831dd7ca85a4a29/.github/workflows/test-e2e-gpu.yaml.

The workflow checks out the untrusted code here:

https://github.com/kubeflow/trainer/blob/d13a831c0072fe10c7e3019aa831dd7ca85a4a29/.github/workflows/test-e2e-gpu.yaml#L37-L42

```
37        - name: Check out code
38          if: steps.check-label.outputs.skip == 'false'
39          uses: actions/checkout@v4
40          with:
41            ref: ${{ github.event.pull_request.head.sha }}
42            path: ${{ env.GOPATH }}/src/github.com/kubeflow/trainer
```

The workflow then proceeds to run commands from the checked out code here:

https://github.com/kubeflow/trainer/blob/d13a831c0072fe10c7e3019aa831dd7ca85a4a29/.github/workflows/test-e2e-gpu.yaml#L62-L71

```
62        - name: Setup cluster with GPU support using nvidia/kind
63          if: steps.check-label.outputs.skip == 'false'
64          run: |
65            make test-e2e-setup-gpu-cluster K8S_VERSION=${{ matrix.kubernetes-version }}
66
67         - name: Run e2e test on GPU cluster
68           if: steps.check-label.outputs.skip == 'false'
69           run: |
70             mkdir -p artifacts/notebooks
71             make test-e2e-notebook NOTEBOOK_INPUT=./examples/torchtune/llama3_2/alpaca-
                   trainjob-yaml.ipynb NOTEBOOK_OUTPUT=./artifacts/notebooks/${{ matrix.
                   kubernetes-version }}_alpaca-trainjob-yaml.ipynb TIMEOUT=900
```

The workflow triggers on `pull_request_target` events. The code has access to the underlying GitHub secret in the workflow, and an attacker can modify their code such that `make test-e2e-notebook` leaks the secret instead of running the e2e test. The workflow does not have any top-level permissions and presumably has write permissions as a result. The combination of these factors can allow an attacker to elevate their privileges to take over the repository.

Code intends to authenticate but does not do so

Severity	Critical
Status	Reported
ID	ADA-KUBEFL-13
Component	Katib and Notebooks

Note, the severity of this finding is critical because it needs follow up and should not be disclosed prior to being scrutinized. The vulnerability was found late in the audit and due to time constrains, the auditing team was not able to complete the assessment due to lack of time. In this disclosure we describe what we have found thus far and we encourage the Kubeflow team to complete the assessment. The severity can change significantly based on the Kubeflow teams own findings.

We found two cases of Kubeflow services that have the goal of authenticating user requests, but where the logic does not actually carry out authentication. The first is the Katib v1beta1 authorization API `IsAuthorized`:

https://github.com/kubeflow/katib/blob/692ee08e7e08f65c210e30ebb10cb5795077b74d/pkg/ui/v1beta1/authzn.go#L44-L72

```
44  func IsAuthorized(verb, namespace, resource, subresource, name string, schema schema.
        GroupVersion, client client.Client, r *http.Request) (string, error) {
45
46      // We disable authn/authz checks when in standalone mode.
47      if DISABLE_AUTH == "true" {
48          log.Printf("APP_DISABLE_AUTH set to True. Skipping authentication/authorization
                checks")
49          return "", nil
50      }
51      // Check if an incoming request is from an authenticated user (kubeflow mode:
            kubeflow-userid header)
52      if r.Header.Get(USER_HEADER) == "" {
53          return "", errors.New("user header not present")
54      }
55      user := r.Header.Get(USER_HEADER)
56      user = strings.Replace(user, USER_PREFIX, "", 1)
57
58      // Check if the user is authorized to perform a given action on katib/k8s resources
            .
59      sar := CreateSAR(user, verb, namespace, resource, subresource, name, schema)
60      err := client.Create(context.TODO(), sar)
61      if err != nil {
62          log.Printf("Error submitting SubjectAccessReview: %v, %s", sar, err.Error())
63          return user, err
64      }
65
66      if sar.Status.Allowed {
67          return user, nil
68      }
69
70      msg := generateUnauthorizedMessage(user, verb, namespace, resource, subresource,
            schema, sar)
71      return user, errors.New(msg)
72  }
```

Here, the comment in question says: "// Check if an incoming request is from an authenticated user (kubeflow mode: kubeflow-userid header)". This is not what the code does. The code does not check whether the request is from an authenticated user,

it merely checks if a header is present in the request. The code does not validate that the user actually corresponds to a valid user or whether the user is authenticated. As such, any request that contains the `USER_HEADER` header with any string is considered authenticated, and anyone who can send requests to the Katib v1beta1 UI is considered authenticated, because they can just do so by including the `USER_HEADER` header in the request.

The second case in this finding is the Kubeflow Notebooks backend:

https://github.com/kubeflow/kubeflow/blob/6dbe516130193616f6304f0464c7d54699792a63/components/crud-web-apps/common/backend/kubeflow/kubeflow/crud_backend/authn.py#L34-L67

```
34  @bp.before_app_request
35  def check_authentication():
36      """
37      By default all the app's routes will be subject to authentication. If we
38      want a function to not have authentication check then we can decorate it
39      with the `no_authentication` decorator.
40      """
41      if config.dev_mode_enabled():
42          log.debug("Skipping authentication check in development mode")
43          return
44
45      if settings.DISABLE_AUTH:
46          log.info("APP_DISABLE_AUTH set to True. Skipping authentication check")
47          return
48
49      # If a function was decorated with `no_authentication` then we will skip
50      # the authn check
51      if request.endpoint and getattr(
52          current_app.view_functions[request.endpoint],
53          "no_authentication",
54          False,
55      ):
56          # when no return value is specified the designated route function will
57          # be called.
58          return
59
60      user = get_username()
61      if user is None:
62          # Return an unauthenticated response and don't call the route's
63          # assigned function.
64          raise Unauthorized("No user detected.")
65      else:
66          log.info("Handling request for user: %s", user)
67          return
```

This is a decorator for all of the apps routes. Its intention, based on reading the code, is to authenticate incoming requests, however, this is not what the code actually does. Also in this case, the code merely checks for the presence of a header in the request which is not authentication. The `get_username()` API does the following:

https://github.com/kubeflow/kubeflow/blob/master/components/crud-web-apps/common/backend/kubeflow/kubeflow/crud_backend/authn.py#L12-L22C20

```
12  def get_username():
13      if settings.USER_HEADER not in request.headers:
14          log.debug("User header not present!")
15          username = None
16      else:
17          user = request.headers[settings.USER_HEADER]
18          username = user.replace(settings.USER_PREFIX, "")
19          log.debug("User: '%s' | Headers: '%s' '%s'",
20                    username, settings.USER_HEADER, settings.USER_PREFIX)
21
```

```
22        return username
```

As such, the logic here does not authenticate, and also here, anyone who can send requests to the server can authenticate, since they can simply send a request that contains the `USER_HEADER` header.

Proper authentication should in this case be made based on at least one piece of sensitive data - such as a token - from which Kubeflow extracts the username and then validates that the username is authenticated.

Server-Side Request Forgery (SSRF) in Kubeflow Pipelines CreatePipelineV1 API

Severity	Moderate
Status	Reported
ID	ADA-KUBEFL-01
Component	Pipelines CreatePipelineV1

Kubeflow Pipelines is vulnerable to an server-side request forgery issue which essentially is a scenario where an untrusted user - which in this case is a user with permissions to create pipelines - makes the `BasePipelineServer` send requests to a URL of their choosing. This can be a security breach if the `BasePipelineServer` has access to services that the untrusted user does not, such as services running on `localhost` or cluster resources that are not exposed to users.

The method that exposes this behavior is `(s *PipelineServer)CreatePipelineAndVersion`:

https://github.com/kubeflow/pipelines/blob/6ccf261e25d5fa0164a783f8b85587dc4f63794f/backend/src/apiserver/server/pipeline_server.go#L628-L651

```
628  func (s *PipelineServer) CreatePipelineAndVersion(ctx context.Context, request *
         apiv2beta1.CreatePipelineAndVersionRequest) (*apiv2beta1.Pipeline, error) {
629      if s.options.CollectMetrics {
630          createPipelineRequests.Inc()
631          createPipelineVersionRequests.Inc()
632      }
633
634      // Convert the input request
635      pipeline, err := toModelPipeline(request.GetPipeline())
636      if err != nil {
637          return nil, util.Wrap(err, "Failed to create a pipeline due to pipeline
                 conversion error")
638      }
639
640      // Create both pipeline and pipeline version in a single transaction
641      newPipeline, _, err := s.createPipelineAndPipelineVersion(ctx, pipeline, request.
             GetPipelineVersion().GetPackageUrl().GetPipelineUrl())
642      if err != nil {
643          return nil, util.Wrap(err, "Failed to create a pipeline")
644      }
645
646      if s.options.CollectMetrics {
647          pipelineCount.Inc()
648          pipelineVersionCount.Inc()
649      }
650      return toApiPipeline(newPipeline), nil
651  }
```

Notice the last parameter passed to `s.createPipelineAndPipelineVersion`. This parameter is a value from the `*apiv2beta1.CreatePipelineAndVersionRequest` which we consider to be untrusted.

If we further examine the dataflow and look at `(s *BasePipelineServer)createPipelineAndPipelineVersion`, we see that Kubeflow Pipelines will send an HTTP request to `pipelineUrl` on line 180:

https://github.com/kubeflow/pipelines/blob/6ccf261e25d5fa0164a783f8b85587dc4f63794f/backend/src/apiserver/server/pipeline_server.go#L144-L200

```
144 func (s *BasePipelineServer) createPipelineAndPipelineVersion(ctx context.Context,
        pipeline *model.Pipeline, pipelineUrlStr string) (*model.Pipeline, *model.
        PipelineVersion, error) {
145     // Resolve name and namespace
146     pipelineFileName := path.Base(pipelineUrlStr)
147 
148     pipeline.Name = buildPipelineName(pipeline.Name, pipeline.DisplayName,
            pipelineFileName)
149     if pipeline.DisplayName == "" {
150         pipeline.DisplayName = pipeline.Name
151     }
152 
153     pipeline.Namespace = s.resourceManager.ReplaceNamespace(pipeline.Namespace)
154 
155     // Check authorization
156     resourceAttributes := &authorizationv1.ResourceAttributes{
157         Namespace: pipeline.Namespace,
158         Name:      pipeline.Name,
159         Verb:      common.RbacResourceVerbCreate,
160     }
161     err := s.canAccessPipeline(ctx, "", resourceAttributes)
162     if err != nil {
163         return nil, nil, err
164     }
165 
166     // Create a pipeline version with the same name and description
167     pipelineVersion := &model.PipelineVersion{
168         Name:            pipeline.Name,
169         DisplayName:     pipeline.DisplayName,
170         PipelineSpecURI: pipelineUrlStr,
171         Description:     pipeline.Description,
172         Status:          model.PipelineVersionCreating,
173     }
174 
175     // Download and parse pipeline spec
176     pipelineUrl, err := url.ParseRequestURI(pipelineUrlStr)
177     if err != nil {
178         return nil, nil, util.NewInvalidInputError("invalid pipeline spec URL: %v",
                pipelineUrlStr)
179     }
180     resp, err := s.httpClient.Get(pipelineUrl.String())
181     if err != nil {
182         return nil, nil, util.NewInternalServerError(err, "error downloading the
                pipeline spec from %v", pipelineUrl.String())
183     } else if resp.StatusCode != http.StatusOK {
184         return nil, nil, util.NewInvalidInputError("error fetching pipeline spec from %
                v - request returned %v", pipelineUrl.String(), resp.Status)
185     }
186     defer resp.Body.Close()
187     pipelineFile, err := ReadPipelineFile(pipelineFileName, resp.Body, common.
            MaxFileLength)
188     if err != nil {
189         return nil, nil, err
190     }
191     pipelineVersion.PipelineSpec = string(pipelineFile)
192 
193     // Validate the pipeline version
194     if err := s.validatePipelineVersionBeforeCreating(pipelineVersion); err != nil {
195         return nil, nil, err
196     }
197 
198     // Create both pipeline and pipeline version is a single transaction
199     return s.resourceManager.CreatePipelineAndPipelineVersion(pipeline, pipelineVersion
```

```
            )
200 }
```

There is no allow-list for domains, no block-list for private/loopback/metadata IPs, no scheme/port restrictions, and no re-validation after redirects. The only checks are URL syntax, HTTP 200 status, a max size, and later spec validation - none of which restrict the network destination.

Note on local files: absolute paths like `/etc/shadow` without a scheme are rejected by `url.ParseRequestURI`, and `file://` URLs are not supported by Go's `http.Client`. The SSRF risk is network-based, not local file read via this code path.

Recommended remediation

Restrict where the backend will fetch from and validate the final destination:

- Replace arbitrary URLs with file uploads or server-generated pre-signed URLs for controlled object storage.
- If URLs must remain, enforce an allow-list of trusted domains; block private/loopback/link-local/metadata IP ranges after DNS resolution and on every redirect; restrict schemes/ports (prefer `https://` on standard ports); and use tight timeouts.

SSRF in Kserve HTTPSProvider

Severity	Moderate
Status	Reported
ID	ADA-KUBEFL-02
Component	Kserve HTTPSProvider

Similar to ADA-KUBEFL-01 in `(h *HTTPSDownloader)Download()` where storageURI is untrusted.

https://github.com/kserve/kserve/blob/15750598b1aab5ca8283e6d304547f4221a9f834/pkg/agent/storage/https.go#L73-L101

```
73  func (h *HTTPSDownloader) Download(client http.Client) error {
74      // Create request
75      req, err := http.NewRequest(http.MethodGet, h.StorageUri, nil)
76      if err != nil {
77          return err
78      }
79
80      headers, err := h.extractHeaders()
81      if err != nil {
82          return err
83      }
84      for key, element := range headers {
85          req.Header.Add(key, element)
86      }
87
88      // Query request
89      resp, err := client.Do(req)
90      if err != nil {
91          return fmt.Errorf("failed to make a request: %w", err)
92      }
93
94      defer func() {
95          if resp.Body != nil {
96              closeErr := resp.Body.Close()
97              if closeErr != nil {
98                  log.Error(closeErr, "failed to close body")
99              }
100         }
101     }()
```

Path Traversal in KServe GCS Downloader

Severity	Moderate
Status	Reported
ID	ADA-KUBEFL-03
Component	Kserve GCS Downloader

When users download models from remote storage, be it cloud registries and even model-registry, there is a possibility that the models may be maliciously configured to an extent where they can cause harm to the user downloading them. In such a case, an attacker somehow places a maliciously configured model in the storage and waits for the victim to download it. When the victim downloads it, and if the model achieves its goal in harming the victim, the attacker has successfully attacked the victim. This kind of attack requires a click from the victim.

Kserve behaves in a way that allows an attacker who can modify models stored in GCS to cause harm to a user that downloads it. The attacker can here be someone who obtains permissions to the GCS which will allow them to escalate privileges and harm users.

The behavior in this disclosure allows the attacker to arbitrarily delete and create files on the victims machine. The root cause of the issue is that Kserve deletes and creates files on the users (victims) machine based on the files it downloads from GCS, so if Kserve downloads a file whose name points to an existing file, Kserve will delete the existing file and replace it with the file from GCS. The danger here comes from the naming of GCS files which can contain “any sequence of valid Unicode characters” (docs) which allows the name to contain path traversing characters.

The vulnerable function is `(g *GCSObjectDownloader)Download()`:

https://github.com/kserve/kserve/blob/15750598b1aab5ca8283e6d304547f4221a9f834/pkg/agent/storage/gcs.go#L81-L119

```
81  func (g *GCSObjectDownloader) Download(ctx context.Context, client stiface.Client, it
        stiface.ObjectIterator) error {
82      var errs []error
83      // flag to help determine if query prefix returned an empty iterator
84      foundObject := false
85
86      for {
87          attrs, err := it.Next()
88          if errors.Is(err, iterator.Done) {
89              break
90          }
91          if err != nil {
92              return fmt.Errorf("an error occurred while iterating: %w", err)
93          }
94          objectValue := strings.TrimPrefix(attrs.Name, g.Item)
95          fileName := filepath.Join(g.ModelDir, g.ModelName, objectValue)
96
97          foundObject = true
98          if FileExists(fileName) {
99              log.Info("Deleting file", "name", fileName)
100             if err := os.Remove(fileName); err != nil {
101                 return fmt.Errorf("file is unable to be deleted: %w", err)
102             }
103         }
104         file, err := Create(fileName)
```

```
105             if err != nil {
106                 return fmt.Errorf("file is already created: %w", err)
107             }
108             if err := g.DownloadFile(ctx, client, attrs, file); err != nil {
109                 errs = append(errs, err)
110             }
111         }
112         if !foundObject {
113             return gstorage.ErrObjectNotExist
114         }
115         if len(errs) > 0 {
116             return awserr.NewBatchError("GCSDownloadIncomplete", "some objects failed to
                download.", errs)
117         }
118         return nil
119     }
```

The `it` argument is an iterator for GCS. `(g *GCSObjectDownloader)Download()` iterates through the objects in that iterator and for each object it deletes it if it exists, creates the file again and downloads the contents to the created file. Here, we consider `attrs.Name` to be untrusted, ie. the Object name in GCS which other users can modify. The Object name can point to a path that on the users system is sensitive. Below, we illustrate that with a unit test where a malicious GCS object overwrites a mocked `etc/passwd` file:

```
1   package storage_test
2
3   import (
4       "context"
5       "io"
6       "os"
7       "path/filepath"
8       "strings"
9       "testing"
10
11      gstorage "cloud.google.com/go/storage"
12      "github.com/kserve/kserve/pkg/agent/mocks"
13      storagepkg "github.com/kserve/kserve/pkg/agent/storage"
14  )
15
16  const originalPasswd = `root:!:0:0::/:/usr/bin/ksh
17  daemon:!:1:1::/etc:
18  bin:!:2:2::/bin:
19  sys:!:3:3::/usr/sys:
20  adm:!:4:4::/var/adm:
21  uucp:!:5:5::/usr/lib/uucp:
22  guest:!:100:100::/home/guest:
23  nobody:!:4294967294:4294967294::/:
24  lpd:!:9:4294967294::/:
25  lp:*:11:11::/var/spool/lp:/bin/false
26  invscout:*:200:1::/var/adm/invscout:/usr/bin/ksh
27  nuucp:*:6:5:uucp login user:/var/spool/uucppublic:/usr/sbin/uucp/uucico
28  paul:!:201:1::/home/paul:/usr/bin/ksh
29  jdoe:*:202:1:John Doe:/home/jdoe:/usr/bin/ksh
30  `
31
32  const modifiedPasswd = `root:!:0:0::/:/usr/bin/ksh
33  daemon:!:1:1::/etc:
34  bin:!:2:2::/bin:
35  sys:!:3:3::/usr/sys:
36  adm:!:4:4::/var/adm:
37  uucp:!:5:5::/usr/lib/uucp:
38  guest:!:100:0::/home/guest:
39  nobody:!:4294967294:4294967294::/:
40  lpd:!:9:4294967294::/:
```

```
lp:*:11:11::/var/spool/lp:/bin/false
invscout:*:200:1::/var/adm/invscout:/usr/bin/ksh
nuucp:*:6:5:uucp login user:/var/spool/lp:/usr/sbin/uucp/uucico
paul:!:201:1::/home/paul:/usr/bin/ksh
jdoe:*:202:1:John Doe:/home/jdoe:/usr/bin/ksh
`

func TestGCSProvider_DownloadModel_PathTraversal_DeletesThenCreates(t *testing.T) {
    t.Parallel()
    ctx := context.Background()
    base := t.TempDir()

    realPasswd := filepath.Join(base, "passwd")
    if err := os.WriteFile(realPasswd, []byte(originalPasswd), 0o600); err != nil {
        t.Fatal(err)
    }

    mockEtc := filepath.Join(base, "etc")
    if err := os.MkdirAll(mockEtc, 0o755); err != nil {
        t.Fatal(err)
    }
    mockEtcPasswd := filepath.Join(mockEtc, "passwd")
    if err := os.WriteFile(mockEtcPasswd, []byte(originalPasswd), 0o644); err != nil {
        t.Fatal(err)
    }

    modelName := "model-subdir"
    if err := os.MkdirAll(filepath.Join(base, modelName), 0o755); err != nil {
        t.Fatal(err)
    }

    // set up KServe GCS mocks
    client := mocks.NewMockClient()

    const bucket = "test-bucket"
    if err := client.Bucket(bucket).Create(ctx, "dummy-project", &gstorage.BucketAttrs
        {}); err != nil {
        t.Fatalf("creating mock bucket: %v", err)
    }

    const prefix = "prefix/"
    const traversalObject = prefix + "../passwd"

    w := client.Bucket(bucket).Object(traversalObject).NewWriter(ctx)
    if _, err := io.WriteString(w, modifiedPasswd); err != nil {
        t.Fatalf("writing mock object: %v", err)
    }

    provider := &storagepkg.GCSProvider{Client: client}
    storageURI := "gs://" + bucket + "/" + prefix

    if err := provider.DownloadModel(base, modelName, storageURI); err != nil {
        t.Fatalf("DownloadModel error: %v", err)
    }

    // Confirm overwrite happened at base/passwd
    got, err := os.ReadFile(realPasswd)
    if err != nil {
        t.Fatal(err)
    }
    if string(got) != modifiedPasswd {
        t.Fatalf("expected modified passwd, got:\n%s", got)
    }
    if !strings.Contains(string(got), "guest:!:100:0::/home/guest:") {
```

```
104             t.Fatalf("expected modified guest GID in passwd")
105         }
106
107         // Ensure NO 'passwd' file was created inside model dir
108         if _, err := os.Stat(filepath.Join(base, modelName, "passwd")); err == nil {
109             t.Fatal("unexpected 'passwd' file inside model dir")
110         }
111
112         // Ensure mock /etc/passwd unchanged
113         mockBytes, _ := os.ReadFile(mockEtcPasswd)
114         if string(mockBytes) != originalPasswd {
115             t.Fatal("mock /etc/passwd unexpectedly modified")
116         }
117     }
```

The unit test changes the guest users group ID on the following line `guest:!:100:100::/home/guest:` to `guest:!:100:0::/home/guest:`.

To run the uni test, place it in kserve/pkg/agent/storage first and run it with `go test -run=TestGCSProvider_DownloadModel_PathT`.

From a quick review, `(s *S3ObjectDownloader)GetAllObjects()` is vulnerable to the same.

Kubeflow Pipelines: MinioObjectStore.GetFile – Reliability & Security Issues

Severity	Moderate
Status	Reported
ID	ADA-KUBEFL-04
Component	Pipelines MinioObjectStore.GetFile

Pipelines's Minio ObjectStore storage implementation can cause a node-wide denial of service. For an attacker to exploit this, they need to be able to modify the objects in the Minio ObjectStore and place a malicious Object and then wait for their victim to download it. At its core, this issue allows anyone with write permissions to the Minio ObjectStore to overwrite all objects and cause denial of service of the node whenever someone downloads any file in Kubeflow Pipelines.

There are two factors that combined make up this issue: 1) Pipelines reads Minio Objects entirely into memory, and 2) Minio Object Store can manage Objects up to 50 TiB of size. As such, a large object can make Pipelines read 50 TiB into memory. We assume that the available memory for most Kubeflow deployments is much lower than that.

The vulnerably API is `(m *MinioObjectStore)GetFile()`:

https://github.com/kubeflow/pipelines/blob/6bf566d87609c56d52287c222efed87a68a71644/backend/src/apiserver/storage/object_store.go#L82-L100

```
82  func (m *MinioObjectStore) GetFile(ctx context.Context, filePath string) ([]byte, error
        ) {
83      reader, err := m.minioClient.GetObject(ctx, m.bucketName, filePath, minio.
            GetObjectOptions{})
84      if err != nil {
85          return nil, util.NewInternalServerError(err, "Failed to get file %v", filePath)
86      }
87
88      buf := new(bytes.Buffer)
89      buf.ReadFrom(reader)
90
91      bytes := buf.Bytes()
92
93      // Remove single part signature if exists
94      if m.disableMultipart {
95          re := regexp.MustCompile(`\w+;chunk-signature=\w+`)
96          bytes = []byte(re.ReplaceAllString(string(bytes), ""))
97      }
98
99      return bytes, nil
100 }
```

This API opens a reader at an object and reads it into a `bytes.Buffer`. It then reads that buffer entirely into memory with the call to `buf.Bytes()` which is the line that will drain the available memory.

Even with the change from minio to seaweedfs and the removal of the dangerous artifact-proxy this problem https://github.com/kubeflow/pipelines/issues/9889 is still valid and orthogonal to the MLMD removal. It is still exploitable also without MLMD. It just completely skips any authorization process for our S3 storage.

References: - MinIO object size thresholds

Pipelines v2 component launcher users vulnerable to path traversal from malicious Object name

Severity	Moderate
Status	Reported
ID	ADA-KUBEFL-07
Component	Pipelines v2

The Pipelines v2 component launcher has a vulnerability that allows an attacker to write files on users' machine if the attacker has permissions to manage the bucket from which the user downloads objects. The reason is that the object storage writes objects to a path determined by the object name, and since blobs - at least in Google Cloud - can have path traversing characters in their names, someone who can modify the blobs in the bucket can escalate their privileges and arbitrarily write files on users' local machines.

The root cause is in `DownloadBlob`: https://github.com/kubeflow/pipelines/blob/7d6722e04b3c95fdc9a7a1250777b404ea2ad80c/backend/src/v2/objectstore/object_store.go#L123-L150

```
123 func DownloadBlob(ctx context.Context, bucket *blob.Bucket, localDir, blobDir string)
        error {
124     iter := bucket.List(&blob.ListOptions{Prefix: blobDir})
125     for {
126         obj, err := iter.Next(ctx)
127         if err != nil {
128             if err == io.EOF {
129                 break
130             }
131             return fmt.Errorf("failed to list objects in remote storage %q: %w",
                    blobDir, err)
132         }
133         if obj.IsDir {
134             // TODO: is this branch possible?
135
136             // Object stores list all files with the same prefix,
137             // there is no need to recursively list each folder.
138             continue
139         } else {
140             relativePath, err := filepath.Rel(blobDir, obj.Key)
141             if err != nil {
142                 return fmt.Errorf("unexpected object key %q when listing %q: %w", obj.
                        Key, blobDir, err)
143             }
144             if err := downloadFile(ctx, bucket, obj.Key, filepath.Join(localDir,
                    relativePath)); err != nil {
145                 return err
146             }
147         }
148     }
149     return nil
150 }
```

`bucket.List()` returns an iterator for blobs in the bucket as per go-clouds docustring:

https://github.com/google/go-cloud/blob/3690eda109e669733ebf10689affa1bcdfeb0cb8/blob/blob.go#L815-L822

“”” // List returns a ListIterator that can be used to iterate over blobs in a // bucket, in lexicographical order of UTF-8 encoded keys. The underlying // implementation fetches results in pages. // // A nil ListOptions is treated the same as the zero value. // // List is not guaranteed to include all recently-written blobs; // some services are only eventually consistent. “””

`DownloadBlob` then iterates over the blobs and invokes `downloadFile` passing both `obj.Key` and a local filepath that is created from `obj.Key`. Here, we understand `obj.Key` can contain path traversing characters.

`downloadFile` then creates the file on the `localFilePath` which is created using the object Key and writes the contents:

https://github.com/kubeflow/pipelines/blob/7d6722e04b3c95fdc9a7a1250777b404ea2ad80c/backend/src/v2/objectstore/object_store.go#L180-L213 ~~~~ {.go .numberLines startFrom=“180”} func downloadFile(ctx context.Context, bucket *blob.Bucket, blobFilePath, localFilePath string) (err error) { errorF := func(err error) error { return fmt.Errorf(“downloadFile(): unable to complete copying %q to local storage %q: %w”, blobFilePath, localFilePath, err) }

```
r, err := bucket.NewReader(ctx, blobFilePath, nil)
if err != nil {
    return errorF(fmt.Errorf("unable to open reader for bucket: %w", err))
}
defer r.Close()

localDir := filepath.Dir(localFilePath)
if err := os.MkdirAll(localDir, 0755); err != nil {
    return errorF(fmt.Errorf("failed to create local directory %q: %w", localDir, err))
}

w, err := os.Create(localFilePath)
if err != nil {
    return errorF(fmt.Errorf("unable to open local file %q for writing: %w",
        localFilePath, err))
}
defer func() {
    errClose := w.Close()
    if err == nil && errClose != nil {
        // override named return value "err" when there's a close error
        err = errorF(errClose)
    }
}()

if _, err = io.Copy(w, r); err != nil {
    return errorF(fmt.Errorf("unable to complete copying: %w", err))
}

return nil
```

} ~~~~~~

The invocation flow from the users perspective looks as such:

1. The user invokes `(l *LauncherV2)Execute()` [ref].
2. The invocation continues into `executeV2` [ref].
3. The invocation continues into `execute` [ref].
4. The invocation continues into `downloadArtifacts` [ref].
5. downloadArtifact loops through the artifacts that the user has specified [ref]. These are the artifacts that the user expects. 6. `downloadArtifacts` downloads each of the artifacts in the loop by calling `objectstore.DownloadBlob` [ref].

To summarise this in a security context, the user invokes `(l *LauncherV2)Execute()` using their parameters. At some point in the process, the launcher downloads files from a remote bucket which can contain path traversing characters.

Because it comes from the remote bucket, other users may be able to modify its name and contents in which case the user would still download it because no filtering occurs.

Resources missing network policies

Severity	Moderate
Status	Reported
ID	ADA-KUBEFL-08
Component	Multiple

Network policies in Kubernetes are useful for assigning granular network permissions for workloads. Lack of a network policy can be a sign of over-privileged workloads since Kubernetes assigns general network permissions by default all of which workloads rarely need. A productive approach for defining network permissions is to assign a network policy without any permissions ('deny all') and then add the permissions that the resource needs. When we carry out in-cluster auditing, we flag resources that have no network policies, as it means they may not have undergone this process of denying all privileges and adding only the required back. When auditing the cluster state we found from installing the manifests with KinD by following the instructions at https://github.com/kubeflow/manifests/blob/c24eef1735f564e463ff09b3f303d47279a26c0c/README.md#install-with-a-single-command, we found the following resources to be missing a network policy:

#	Name	Kind	Namespace
1	dex	Deployment	auth
2	cert-manager	Deployment	cert-manager
3	cert-manager-cainjector	Deployment	cert-manager
4	cert-manager-webhook	Deployment	cert-manager
5	kube-hunter	Job	default
6	kube-apiserver-kubeflow-control-plane	Pod	kube-system
7	istio-cni-node	DaemonSet	kube-system
8	kindnet	DaemonSet	kube-system
9	istiod	Deployment	istio-system
10	cluster-local-gateway	Deployment	istio-system
11	istio-ingressgateway	Deployment	istio-system
12	local-path-provisioner	Deployment	local-path-storage
13	oauth2-proxy	Deployment	oauth2-proxy
14	webhook	Deployment	knative-serving
15	net-istio-webhook	Deployment	knative-serving
16	controller	Deployment	knative-serving
17	autoscaler	Deployment	knative-serving
18	net-istio-controller	Deployment	knative-serving
19	activator	Deployment	knative-serving

Model-Registry enables profiling endpoints by default

Severity	Moderate
Status	Fixed
ID	ADA-KUBEFL-12
Component	Model Registry

Model-Registry exposes the Go profiling interface at `/debug/pprof`. This endpoint is designed for performance diagnostics, but when left accessible over the public internet, it creates a significant security risk.

https://github.com/kubeflow/model-registry/blob/55baaff414cf693b2d31d3790819643e6fe5f359/main.go#L8

```
1  package main
2
3  import (
4      "github.com/golang/glog"
5      "github.com/kubeflow/model-registry/cmd"
6      "log"
7      "net/http"
8      _ "net/http/pprof"
9  )
```

The profiling interface reveals detailed runtime information about the application, including stack traces, goroutines, heap allocations, and CPU usage. In the hands of an attacker, this information can be used to:

- Gain deep insights into the internal structure and operation of the application.
- Enumerate dependencies, internal functions, and error conditions that may assist in developing targeted exploits.
- Cause denial of service by triggering resource-intensive profiling operations such as heap or trace dumps.

Although the endpoint does not directly provide code execution, its value as an information disclosure vector and reconnaissance tool should not be underestimated. In modern attack chains, such information often makes the difference between a blocked intrusion and a successful compromise.

Similar exposures have been observed in the wild:

1. **Kubernetes CVE-2019-11248**: Profiling endpoints were left accessible through the Kubelet's health port, leading to unauthenticated information disclosure and the potential for denial-of-service attacks (Rapid7 research).
2. **Prometheus exposures**: Research has shown that over 300,000 Prometheus servers and exporters had debug/pprof endpoints exposed, putting them at risk of profiling-based denial of service and leaking sensitive runtime data (Aqua Security research).

These cases demonstrate that attackers actively scan for such endpoints and use them to prepare or amplify further attacks. To reduce the attack surface, the `/debug/pprof` interface should not be exposed in production environments. Best practice is to disable it entirely by default, and only enable it when explicitly needed for troubleshooting, with proper safeguards such as authentication, IP whitelisting, or access through a private network. Disable the endpoints by removing the import.

Katib uses weak image reference validation

Severity	Low
Status	Reported
ID	ADA-KUBEFL-05
Component	Katib

Katib uses weak image reference validation which does not require the reference to be fully specified thereby potentially making it vulnerable to several supply-chain attacks: https://github.com/kubeflow/katib/blob/fe7a35dffa2400e2cdfe106452df17468ed75185/pkg/webhook/v1beta1/pod/utils.go#L66-L94

```
66  func getRemoteImage(pod *v1.Pod, namespace string, containerIndex int) (crv1.Image,
        error) {
67      // verify the image name, then download the remote config file
68      c := pod.Spec.Containers[containerIndex]
69      ref, err := name.ParseReference(c.Image, name.WeakValidation)
70      if err != nil {
71          return nil, fmt.Errorf("Failed to parse image %q: %v", c.Image, err)
72      }
73      imagePullSecrets := []string{}
74      for _, s := range pod.Spec.ImagePullSecrets {
75          imagePullSecrets = append(imagePullSecrets, s.Name)
76      }
77      kc, err := k8schain.NewInCluster(context.TODO(),
78          k8schain.Options{
79              Namespace:          namespace,
80              ServiceAccountName: pod.Spec.ServiceAccountName,
81              ImagePullSecrets:   imagePullSecrets,
82          })
83      if err != nil {
84          return nil, fmt.Errorf("Failed to create k8schain: %v", err)
85      }
86
87      mkc := authn.NewMultiKeychain(kc)
88      img, err := remote.Image(ref, remote.WithAuthFromKeychain(mkc))
89      if err != nil {
90          return nil, fmt.Errorf("Failed to get container image %q info from registry: %v
                ", c.Image, err)
91      }
92
93      return img, nil
94  }
```

We recommend switching to strict validation to require secure image referencing. If that is not possible to enable by default, we recommend allowing users to configure Katib to require strict validation.

Possible DoS from malicious tar archive

Severity	Low
Status	Reported
ID	ADA-KUBEFL-06
Component	Pipelines backend

The `ExtractTgz` utility function which unpacks a tar archive can cause denial of service by consuming all memory of the node if the tar archive contains a sufficiently large file. `ExtractTgz` reads each file entirely into memory on line 79, and if the file is sufficiently large, Pipelines can consume all of the memory on the node:

https://github.com/kubeflow/pipelines/blob/6ccf261e25d5fa0164a783f8b85587dc4f63794f/backend/src/common/util/tgz.go#L59-L86

```
59  func ExtractTgz(tgzContent string) (map[string]string, error) {
60      sr := strings.NewReader(tgzContent)
61      gr, err := gzip.NewReader(sr)
62      if err != nil {
63          return nil, err
64      }
65      tr := tar.NewReader(gr)
66
67      files := make(map[string]string)
68      for {
69          hdr, err := tr.Next()
70          if err == io.EOF {
71              break
72          }
73          if err != nil {
74              return nil, err
75          }
76          if hdr == nil {
77              continue
78          }
79          fileContent, err := io.ReadAll(tr)
80          if err != nil {
81              return nil, err
82          }
83          files[hdr.Name] = string(fileContent)
84      }
85      return files, nil
86  }
```

Pipelines uses this utility function when fetching an artifact:

https://github.com/kubeflow/pipelines/blob/7d6722e04b3c95fdc9a7a1250777b404ea2ad80c/backend/src/common/util/workflow.go#L585-L593

```
585      artifactResponse, err := retrieveArtifact(artifactRequest)
586      if err != nil {
587          return "", err
588      }
589      if artifactResponse == nil || artifactResponse.GetData() == nil || len(
             artifactResponse.GetData()) == 0 {
590          // If artifact is not found or empty content, skip the reporting.
```

```
591             return "", nil
592         }
593         archivedFiles, err := ExtractTgz(string(artifactResponse.GetData()))
```

As such, for an attacker to exploit this, they would need to replace the artifact and wait for the victim to download it. In that moment, the node running Pipelines could be unavailable to all users in the cluster.

Private key included in source code

Severity	Informational
Status	Resolved
ID	ADA-KUBEFL-11
Component	Model Registry

We have identified what appears to be a private key committed to the repository's history. The key is present in the following file and commit:

- **File**: internal/tls/config_test.go
- **Commit**: fd122ba213a56d5066dde40b54cf331fcf257717
- **URL**: https://github.com/kubeflow/model-registry/blob/fd122ba213a56d5066dde40b54cf331fcf257717/internal/tls/config_test.go#L37-L64
- **Relevant section**: Lines 37–64

If this key was accidentally pushed and corresponds to a real certificate or system, its presence constitutes a security issue under CWE-200: Information Exposure. Private keys are considered highly sensitive, and leaks can enable impersonation or unauthorized access if the key remains valid. It is possible that this key was included only for testing purposes or is no longer in use, in which case the exposure is less critical. However, we recommend verifying whether the key is sensitive and still valid. If it is, access should be revoked and the key rotated immediately. Even if the key is not in use, it may be beneficial to remove or replace it with a non-sensitive placeholder to avoid confusion for future contributors and to prevent potential misuse.

Stack-overflow bugs in 3rd-party dependency

Severity	Informational
Status	Reported
ID	ADA-KUBEFL-14
Component	Dependency to Model Registry

The fuzzer we wrote for Model Registry found 3 stack-overflow bugs in a 3rd-party dependency. Stack-overflow crashes are not recoverable in Go which is what Model Registry is implemented in, and as such they as a bug class has the potential to cause denial of service. In this particular case, the input that causes the crash is trusted and does not constitute a security issue. Furthermore, the input is not meant to change often, and if a higher privileged user by mistake configures Model Registry to process such input, they will likely detect the bug quickly.

The three crashes are tracked here:

1. https://issues.oss-fuzz.com/issues/444908412
2. https://issues.oss-fuzz.com/issues/444775605
3. https://issues.oss-fuzz.com/issues/445762259

Users on this list have permissions to view details about the crashes: https://github.com/google/oss-fuzz/blob/0e7214594967696920e84856635f3d0a97f98ed2/projects/kubeflow-model-registry/project.yaml#L5.