lagopus

Release 0.0.1

Sep 23, 2021

Contents:

Lagopus is a distributed fuzzing platform. It allows you to run multiple fuzzing jobs across multiple machines.

CHAPTER 1

About

Lagopus is a distributed fuzzing platform. It allows you to run multiple fuzzing jobs across multiple machines.

Lagopus handles all lifecycle management for fuzzing workloads, including creating, distributing, running, and monitoring fuzzing jobs. It automatically analyzes crashes, minimizes test cases, and manages corpuses. It supports libFuzzer and AFLplusplus out of the box, but can be made to support any fuzzing driver or framework.

Lagopus intends to be an alternative to ClusterFuzz with a focus on a more modular codebase, better hackability and first class support for on-prem clusters and single-node deployments.

See also:

Installation

1.1 Architecture

Lagopus is built on Kubernetes (k8s). The core application runs as a set of k8s containers. Fuzzing jobs run in additional containers created by the core. Kubernetes handles cluster and resource management, job distribution, container lifecycle, and to some extent storage. Lagopus has four main components, each corresponding to one container image. There is one instance of each image in a Lagopus deployment, except for fuzzing containers, which are created on demand to run jobs.

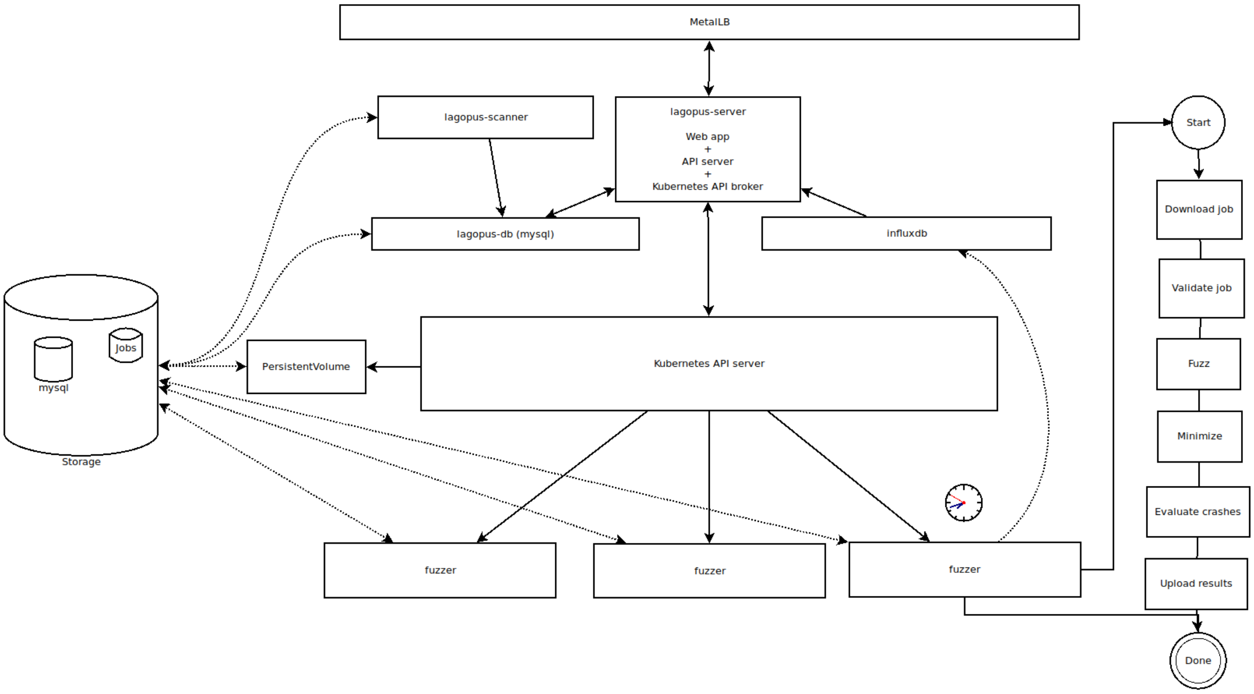
The first component is `lagopus-server`. This is more or less the application core. It is a Flask app that implements the REST API used to interact with Lagopus. It talks to the k8s API to manage cluster resources, primarily to spin up containers for running fuzzing jobs. It is stateless; application state is stored in `lagopus-db`.

The second is `lagopus-db`, which is just a containerized MySQL instance that provides the application database. Details on jobs, crashes, corpuses, etc. are all stored here.

The third is `lagopus-scanner`. When fuzzing jobs complete, they dump their artifacts - minimized corpuses, crashing inputs, and logs - to the Lagopus shared storage area for later use. This container periodically scans that directory looking for recently finished jobs in order to post-process them and import their results into the database. This container is also stateless, and just runs a Python script that does the importing.

The fourth is `lagopus-fuzzer`. This is an Ubuntu 18.04 container image preloaded with a collection of fuzzing utilities. Each fuzzing job is run in a new instance of this image. In the future, support for custom containers should allow a choice of platforms.

Here's a diagram that probably won't make much sense, but at least provides some overview of how the pieces fit together:



1.1.1 Why Kubernetes?

Kubernetes was chosen not out of any particular desire to use microservices, but because it provides both container management and a distributed systems platform, both of which Lagopus needs. It was decided early on that Lagopus should not try to roll its own versions of these two things.

Unfortunately, k8s has something of a reputation for being very complex and unwieldy, and to some extent this is true. It does much more than Lagopus needs it to do. Fortunately the k8s setup required to run Lagopus is relatively minimal; a cluster, some sysctls on the nodes, and an NFS volume.

CHAPTER 2

Installation

How to install Lagopus!

Note: Installing Lagopus is rather difficult right now, since it's still very much a work in progress. You will probably have a hard time unless you already have some operational experience with Kubernetes. This setup process will be improved prior to the initial release to make it easier and more accessible.

The installation process for Lagopus is roughly:

1. Set up a Kubernetes cluster
2. Configure the cluster nodes; some `sysctl`'s need to be set on the nodes for performance reasons, and k8s doesn't have the ability to do that itself right now. The necessary changes can be done with Ansible to make it easier.
3. Create an NFS share accessible by the cluster
4. Clone the Lagopus repository
5. Run `helm install charts/lagopus`

Presently, the Docker images are stored on my personal Docker Hub instance, but those will be moved to something more offical before the initial release.

2.1 Guide

The steps below assume you are using Ubuntu 18.04 LTS on your cluster nodes. More generic instructions should be available prior to the initial release.

2.1.1 NFS

lagopus uses NFS as its storage system. This allows you to keep lagopus storage on any device you want; it doesn't even have to be on a cluster node. As long as the NFS server is accessible from the cluster you can use it.

This section describes how to set up an NFS share on Ubuntu 18.04. If you want to use some other system, that's fine; there are lots of tutorials on how to set up NFS shares online, it's pretty easy.

- Pick somewhere to host NFS on - the master node is okay for this and usually easiest, but any cluster-accessible machine will work.

  > **Warning:** This node should have **lots** of disk space, at least 200gb for production deployments; more depending on how heavy your usage is. Presently Lagopus doesn't do any management of disk resources itself, which is a known limitation; for now, just give yourself as much storage headroom as you can. If you're just trying it out, 10gb or so should be sufficient depending on your job sizes.

- Install NFS:

  ```
  sudo apt update && sudo apt install -y nfs-kernel-server
  ```

- Make a share directory:

  ```
  sudo mkdir -p /opt/lagopus_storage
  sudo chown nobody:nogroup /opt/lagopus_storage
  ```

- Export this share to NFS:

  ```
  echo "/opt/lagopus_storage *(rw,sync,no_subtree_check,no_root_squash)" >> /etc/
  ↪exports
  systemctl restart nfs-server
  ```

- Open firewall to allow NFS, if necessary
- Verify that NFS is working by trying to access it from a cluster node:

  ```
  apt install -y nfs-common && showmount -e <nfs_host>
  ```

  If it's working, you should see:

  ```
  Export list for <nfs_host>:
  /opt/lagopus_storage ::
  ```

Take note of the hostname or IP address of the NFS server, and the share path. You will need to specify them when installing lagopus.

2.1.2 Cluster Configuration

This section is broken down by platform. Each k8s implementation has its quirks. If you're setting up a new cluster I recommend k3s. If you want to test locally I recommend kind or minikube. microk8s. is also an acceptable choice, but you have to deal with snaps, which have many problems. Don't use microk8s if you have ZFS anywhere in your cluster, your troubles will be endless.

Basic node setup

This section assumes you already have a cluster. It is agnostic to whatever implementation of k8s you choose.

Each node in the cluster needs a few tweaks to support lagopus. The necessary changes are:

- Install NFS support

- Normalize core dumps
- Disable apport (Ubuntu only)
- Disable swap
- Allow the kubelet to provision static cpu resources (`--cpu-manager-policy=static`)
- Set kernel CPU scheduler to performance mode

The last 3 are required for AFL to work as a fuzzing driver.

On each node, do the following:

1. Install NFS support

   This is OS-dependent. For example, on Ubuntu:

   ```
   apt update
   apt install -y nfs-common
   ```

2. Normalize core dumps:

   ```
   echo "kernel.core_pattern=core" >> /etc/sysctl.conf
   sysctl -p
   ```

3. If on Ubuntu, the previous setting will be overwritten by Apport each boot. You need to disable Apport:

   ```
   systemctl stop apport
   systemctl disable apport
   ```

4. Next, disable swap to prevent fuzzer memory from being swapped, which hurts performance:

   ```
   swapoff -a
   ```

5. Set the CPU governor to `performance`:

   ```
   cd /sys/devices/system/cpu; echo performance | tee cpu*/cpufreq/scaling_governor
   ```

6. Set the following kubelet parameters on each of your nodes and restart kubelet:

   ```
   --cpu-manager-policy=static
   --kube-reserved="cpu=200m,memory=512Mi"
   ```

   The first option is absolutely necessary to allow fuzzing jobs to bind to CPUs (required by `AFLplusplus`). The second one reserves some resources for the `kubelet` process itself, so that fuzzing jobs cannot starve `kubelet`.

   - microk8s:

     Add the above lines to `/var/snap/microk8s/current/args/kubelet`, then run the following to apply them immediately:

     ```
     rm /var/snap/microk8s/common/var/lib/kubelet/cpu_manager_state
     systemctl reset-failed snap.microk8s.daemon-kubelet
     systemctl restart snap.microk8s.daemon-kubelet
     ```

     If the service fails, check `journalctl -u snap.microk8s.daemon-kubelet` for debugging logs.

On the master node (or the host when using `kind`) you need to install Helm. Lagopus is packaged as a Helm Chart, so you need Helm to install it.

Installing helm is easy; go here, download the latest 3.x release for your platform, extract the tarball and put the `helm` binary in `/usr/local/bin`. If necessary, `chmod +x /usr/local/bin/helm`.

kind

kind is a nice option for running locally without needing a physical cluster. `kind` spins up a cluster on your local machine by running k8s inside of docker. It's oriented towards proof-of-concept and local deployments.

Follow the instructions on the `kind` homepage to install kind and create a cluster. After creating a cluster, go through the steps in *Basic node setup*.

In `kind`, you can log into the nodes as you would a docker container. Find the container IDs of the cluster nodes with `docker ps`:

```
qlyoung@host ~> docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED          ␣
↪   STATUS              PORTS                       NAMES
98bae8548619        kindest/node:v1.18.2   "/usr/local/bin/entr..."   2 hours ago    ␣
↪     Up 2 hours          127.0.0.1:39245->6443/tcp   kind-control-plane
```

After running through the *Basic node setup*, you need to get the LAN IP of the `kind` master node. This is the IP that lagopus will expose its web interface on. Log into the master node, then:

```
ip addr show eth0
```

It should be the first address. For example, on my `kind` cluster:

```
# ip addr show eth0
30: eth0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP␣
↪group default
    link/ether 02:42:ac:13:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.19.0.2/16 brd 172.19.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fc00:f853:ccd:e793::2/64 scope global nodad
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe13:2/64 scope link
       valid_lft forever preferred_lft forever
```

The address is `172.19.0.2`. You should verify that this address is reachable from your host by pinging it. Note this address; this is what you'll use as `lagopusIP` when installing lagopus.

At this point you can skip to *Installing*.

k3s

Go through the steps in *Basic node setup*.

TODO: document how to enable static CPU scheduling for k3s kubelets

microk8s

If you already have a cluster set up, here is an Ansible playbook to do all of the steps described if your nodes are running microk8s on Ubuntu 18.04. Change `qlyoung` to any root-privileged account.

```
- hosts: fuzzers
  vars:
    fuzzing_user: qlyoung
  remote_user: {{ fuzzing_user }}
  become: yes
  become_method: sudo
  gather_facts: no
  pre_tasks:
    - name: 'install python2'
      raw: sudo apt-get -y install python
  tasks:
  - name: install-microk8s
    command: snap install microk8s --classic
  - name: microk8s-perms
    command: sudo usermod -a -G microk8s {{ fuzzing_user }}
  - name: microk8s-enable-dns
    command: microk8s.enable dns
  - name: disable-apport
    shell: |
      systemctl disable apport
      systemctl stop apport
    ignore_errors: yes
  - name: set-kernel-core-pattern
    shell: echo 'kernel.core_pattern=core' >> /etc/sysctl.conf && sysctl -p
  - name: set-kubelet-resources
    shell: |
      echo '--cpu-manager-policy=static' >> /var/snap/microk8s/current/args/kubelet
      echo '--kube-reserved="cpu=200m,memory=512Mi"' >> /var/snap/microk8s/current/
↪args/kubelet
      rm /var/snap/microk8s/common/var/lib/kubelet/cpu_manager_state
      systemctl reset-failed snap.microk8s.daemon-kubelet
      systemctl restart snap.microk8s.daemon-kubelet
  - name: install-nfs
    command: apt install -y nfs-common
  - name: set-kernel-scheduler-performance
    command: cd /sys/devices/system/cpu; echo performance | tee cpu*/cpufreq/scaling_
↪governor
    ignore_errors: yes
```

If the service fails, check `journalctl -u snap.microk8s.daemon-kubelet` for debugging logs.

2.1.3 Building

This is for development purposes, you do not need to do this if you just want to deploy the latest release.

`cd` into the repository. Make your changes. Open `build.sh` and edit the repository information to point at your own Docker repository. Then run `build.sh` to build and push the images.

After that you need to replace all the hardcoded references to my repo in the Helm templates with yours (look for `qlyoung` in `chart/lagopus/templates`).

2.1.4 Installing

To install Lagopus onto the cluster, clone the repository, `cd` into it, then:

```
helm install --set lagopusStorageServer=<nfs_host>,lagopusStoragePath=<nfs_share_path>
↪,lagopusIP=<prefix> <release_name> ./chart/lagopus
```

where:

- `nfs_host` is the hostname of your nfs server
- `nfs_share_path` is the path of the share you want lagopus to use as its storage
- `prefix` is an address range from which to select the IP address to host the lagopus web interface and API on. If you want to use a specific address, pass it as a /32 prefix (e.g. `1.2.3.4/32`). This address should be directly connected relative to the external cluster network; for instance, if your cluster machines have addresses in `172.19.0.0/24`, a reasonable choice might be `172.19.0.2/32`. In practice, you probably want to use the "public" IP of the master k8s node.

Lagopus will select one of the IPs out of the range you configured during installation to expose the web interface. To get this address:

```
kubectl get service | grep lagopus-server | tr -s ' ' | cut -d' ' -f4
```

Supposing the IP address is `A.B.C.D`, you can access the web interface by navigating to http://A.B.C.D/ in your browser. Lagopus does not yet support TLS.

2.1.5 Uninstalling

To remove Lagopus from the cluster, uninstall it with Helm.

::

```
helm uninstall charts/lagopus
```

CHAPTER 3

Usage

Note: TODO: add screenshots

The Lagopus user interface is exposed as a web application. It is served on port 80 at the IP address you configured during the *installation process*. There are several components, each available on a different part of the web interface.

The following pages are each linked in the sidebar of the web interface.

3.1 Dashboard

The dashboard is the home page of the web interface. It has a list of jobs that have been submitted to Lagopus. This list is sorted by recency and includes both running and completed (or failed) jobs. Each job name in this list is linked to the details page for that job, which contains monitoring and statistics information, a summary of fuzzing results, and some controls (presently just a “Kill” button to stop the job).

At the top of the page there are a few cards containing summary information about the Lagopus deployment, including the current number of online nodes as reported by Kubernetes and the number of running jobs.

To the right of the summary information is the “New Job” button that is used to create new fuzzing jobs.

3.2 Jobs

Lagopus is designed around the concept of a `Job`. A job is an individual fuzzing session. Associated with a job are resources such as the containers used to run it, the input zip defining it, its results, its location on the storage volume, and so on.

Each Job has a page in the interface that provides all information about it. This includes its current status, fine-grained statistics, progress information for running jobs, a table of any discovered crashes, code coverage information (not yet implemented), its resource limits, and what node it is running on. This page is accessible by clicking on the name of the job from the Dashboard.

3.2.1 Creating Jobs

Lagopus accepts job definitions in a format very similar to ClusterFuzz. It wants a zip archive with the following structure:

```
job.zip
├── corpus
├── provision.sh
├── target
└── target.conf
```

Where:

- `corpus` is a directory containing a fuzzing corpus; it may be empty, but must be present
- `provision.sh` is a provisioning script used to setup the environment for the target (more on this below)
- `target` is your target binary
- `target.conf` is a config file for afl-multicore; this is only necessary when the job type is *afl*. libFuzzer jobs do not use this.

In addition to these files, you can include anything else you want in this zip archive. This allows you to include e.g. config files or shared libraries needed by the target.

The provision script allows you to customize the container used to run the job. It will probably be necessary for most targets. This script is run before any fuzzing takes place. Use it to install config files, packages, shared libraries and anything else needed to run the target. Remember that the fuzzing container is an Ubuntu 18.04 image, so you have access to all of Ubuntu's apt repositories and can safely install any packages you need. Set it up however you want; if you want to download some file, build some program from source, delete system directories, whatever you want, feel free. Just don't delete `/workdir`, `/<jobname>`, or any of the fuzzing tools ;).

3.3 Crashes

The main goal of any fuzzing system is to find bugs in target programs. When a fuzzing job finds a crash, Lagopus automatically collects information about the crash and imports it into its crash database. The contents of this database are accessible via the Crashes page.

Each entry in the database contains the name of the job it is associated with and the exit code of the target when run with the crashing input. Lagopus also tries to describe the type of the crash by looking at the output of the program when run with the crashing input. For example, Lagopus understands ASAN/MSAN/TSAN/UBSAN output and will store the crash type reported by the sanitizer (e.g. buffer overflow, race condition, etc.) in the `Type` field.

Note: Crash analysis is performed with slightly modified code lifted from ClusterFuzz, so credit goes to Google for that piece.

The output of the program when run with the crashing input is available in the `Backtrace` column.

Each crash table entry also has a link to the fuzzing input that caused the crash in the "Sample" column. Clicking this link downloads the input. This is useful for local debugging.

Note: Depending on the target and fuzzer, the backtrace may show a successful run and the sample provided for download may not reproduce the crash. This typically occurs with libFuzzer targets that accumulate state; the crash may only reproduce when 100 inputs are run in a particular sequence, building up the state necessary to create the error condition within the target. After finding a crashing input, Lagopus attempts to re-run the target with the input to generate a clean backtrace for analysis. If it doesn't cause a crash, Lagopus will fall back to scraping the job logs to get the backtrace, if available. When this happens, the exit code is logged as 101.

If you want to see crashes only for a particular job, go to that job's page and click the "Crashes" tab.

3.4 API

Lagopus exposes an HTTP REST API. The web interface controls Lagopus solely through this API to ensure that it stays up to date and covers all public functionality. The `API` link in the sidebar brings up Swagger-generated API docs. Each endpoint has a documentation blurb associated with it that explains the purpose and usage of the endpoint.

The API provides programmatic access to any task achievable via the web interface.

Because Lagopus itself has no facilities for recurring jobs, CI integration, email reporting, and other desirable features, the goal of the API is to allow as much flexibility and extensibility as possible. For instance, if you want to kick off a fuzz job after each build of your project in CI, you can simply build a job zip as one of your CI artifacts and POST it to the job creation endpoint.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search