Clark Bains
COMP 4000
November 2, 2021
101149052

1 The Project

Back in August, Instructor Dave Mckenney reached out to me, knowing I was interested in server maintenance and deployment, and asked if I would be willing to manage the openstack project for the Carleton Web Development Club (CWDC) that he runs, and that I had worked with on a few occasions. I agreed to do it, and we agreed on a few principles for how it should be set up.

- I should easily be able to allocate resources for students, and give them an environment to run their code in.
- It should be simple to get started on. Not everyone will have extensive Linux experience.
- It should be somewhat secure
- It has to work for students outside of the School of Compute Science.
- It has to run on Openstack

When I first started, we were given a fairly odd allocation of resources, with the major constraint being only 8gb of storage on each of the 6 cluster nodes. This presented a lot of challenges, and not knowing much better, I setup a manually administrated environment, and wrote many scripts to help students do what they needed to do. It wasn't till I met with Forest Anderson, the sysadmin for the Carleton Computer Science Society (CCSS), and saw what they were doing that I figured out that there was a far better way to do it. I then migrated all the existing services to the CCSS cluster, obtained far more Openstack resources, and started taking time to overhaul the CWDC cluster. I added the following requirements:

- Should be easy to provision new servers
- Security should be a far higher priority
- Students shouldn't have to do *any* scripting to get up and running
- Some systems should be redundant (Storage, cluster management)

2 The Design

2.1 Provisioning

Starting from the ground up, I wanted to be able to test configurations locally, using Vagrant. Vagrant would allow me to hook into libvirt on my local system, and rapidly spin up, destroy, and start ansible to provision VMs.

Ansible would let me write configuration files to run specific jobs on the active hosts. For example, I could write files to distribute /etc/hosts files, reboot, install software, and many other different tasks.

2.2 Service Discovery + Distributed metadata store

To achieve my redundancy goal, I would need a common framework to help connect together all of the nodes. I chose Hashicorp's Consul to do this, as it integrated nicely with other software in the stack. Consul helps connect a few master (server) nodes to many client nodes, and allows programatic registration, health check definitions, service querying, and a small distributed data store to be accessible from all the Consul servers and clients.

2.3 Configuration + Secret Store

While I don't anticipate it being used lots, I wanted to provide a distributed, encrypted data store. I used Hashicorp's Vault for this, it can run in a HA mode, storing data on Consul (which is also HA). By registering itself with Consul, other services can use Consul's service lookup (works via DNS queries, e.g https://active.vault.service.consul) to find active Vault servers to interact with.

2.3.1 Vault Transit Store - Auto Unsealing

Since Vault encrypts all data, it requires a number of keys to unseal it before it can respond to any requests. When using HA, this unseal process must happen on each vault HA node. A production system should see these keys distributed among trusted individuals, so especially during the production-but-still-testing state, I wanted to mitigate the risk of sealing the vault (ie for a minor config change) and having to wait for the threshold number of keyholders to be available to unseal all HA vault nodes again. This would also require giving access to the Vault servers to multiple people, which isn't great from a security standpoint. To work around this, I wanted to use a second vault instance, not in HA, to store the encryption keys required to unseal the primary Vault. As long as the secondary Vault transit store stays unsealed, changes can be made to the primary HA vault cluster without requiring the threshold number of keyholders.

2.4 Nomad + Docker

Nomad serves as the job schedule and runner. Everything else is just supporting infrastructure. Nomad takes jobs, and works much like docker-compose. You can specify information about the job, and submit it to one of the server nodes. Nomad will then allocate the job on one of the Nomad clients. Nomad allows for various kinds of execution environments, such as exec directly on the client, JVM, or docker. I wanted to use docker because of the enhanced security and isolation, as well as the more consistent environment that CWDC members can experiment with, locally or on the cluster. Since Nomad is also a Hashicorp software, it integrates well into the stack, registering services, and their health check, on consul, automatically. It can also work with vault, and have a templated job configuration file that injects secrets from vault into the job's environment, allowing members to upload their job specification files online and not worry about secrets. Given that Nomad has far less popularity than many other orchestration solutions, making it as easy as possible for members to share their configs will help students with similar needs to get their project off the ground.

Nomad job files can specify multiple instances (allocations), and combined with consul service discovery and health checks, other services can always discover a healthy allocation of a job and route traffic to it, assuming one does exist.

2.5 Mass Data Store

Since Docker was chosen as the Nomad execution environment, and Docker uses ephemeral containers, I needed to have some form of networked, persistent, redundant data store. For this, I chose Ceph. Ceph uses multiple storage controllers, and storage daemons, each writing data to their own disk. With only one Openstack networked volume to attach, we would have a single point of failure. By using loopback devices, the data gets distributed over multiple servers, and will be resilient to a single host failure. Nomad + Docker can access storage devices using the Container Storage Interface, which Ceph has. This allows fine grained ACLs and proper configuration for each volume mounted to a nomad job instance, and makes it more host agnostic compared to docker bind mounted FUSE mounts. Mounting this way also means that if a Nomad client ever needs all jobs to be evacuated, the jobs will still have access over the network to Ceph.

2.6 Reverse Proxy Ingress

Since the CWDC is a Web Development club after all, most services will be exposing some form of http service. By running the Traefik reverse proxy, and connecting it to consul, Traefik can discover the active and healthy nomad jobs (which will mostly be student projects), and round-robin route traffic to the correct job's

allocation. Traefik also allows for generating trusted SSL certs, using letsencrypt, enhancing the security of user applications, with minimal student effort.

2.6.1 User Submitted Jobs

Since one of the goals of this project is to allow students to easily deploy their own project, I chose to use Hashicorp Waypoint. It allows users, with a single static binary, to login to a waypoint server, and deploy their code. It has the ability to generate docker images of projects, even without a Dockerfile, reducing the learning curve to use it. In one configuration file, you can specify information like variables, which container registry to push to, and the target platform to deploy on. When deploying on Nomad, you can also include the job specification file, so all configuration for a specific project to be run can be included this way. Waypoint, if you update your projects git repository, can be used to start remote builders on the cluster, so you can build docker images for nomad without needing docker, or you can do it locally if you have the compute power and internet bandwidth.

2.7 Security

All services with ACLs should have them enabled, and fairly restrictive. Once a user gets access to the cluster, the design of the system should involve very little manual oversight from me, so it has to be pretty secure.

As an example, I could imagine a insecure Consul instance being connected to, and a user could redefine the service for the club site API, and Traefik would pick it up and have all requests (including authentication), go to the attackers own servers, and they could steal credentials. While I think this is extremely unlikely, adding ACLs makes the cluster more secure against attacks like this.

To protect against users who decide to be malicious, or get hacked, I would like to implement a few mitigations.

- ACLs on all services that support them
- System, core, and user "Zones", so that more trusted, core services (like the club API and frontend) don't rely on the same instance of services that could otherwise be modified by a service in the user zone.
- Internal services and dashboards should either only be accessible via SSH tcp forwarding, alternatively protected in some other way and at most, only be accessible from within the Carleton network.

Since the entire group runs via a Discord server, I plan to write an auth proxy, where any requests going into traefik services with it enabled, will be forwarded to my service. My auth proxy could then ask for a discord name/id, and then send some verification link over discord, and send a message to the auth proxy front end to forward the user to the correct page. This way, user management for administrative dashboard could be as simple as discord roles.

3 Current Implementation

I have spent many late nights working on this, so I am mostly done. All that remains is fine tuning. I have the configuration to wipe Openstack, provision the cluster, have my primary vault unseal using my secondary vault, run a waypoint job, on nomad (which relies on consul), and have it access a volume mounted from ceph through CSI. It would then get an automated SSL certificate and be available on the public internet through traefik.

In my current provisioning implementation, I have nearly 1000 lines of ansible configuration, representing 156 different named tasks (and likely a few more I neglected to name) 700 lines of nomad/vault/consul HCL configuration, with additional scripts to generate a /etc/hosts file based off of an openstack project, and to reset all existing instances.

In addition I have many configs that I used to use, but then changed to be more flexible, more organized, or to use built in functionality rather than custom building it.

3.1 Provisioning

Currently, I have an ansible playbook to setup most of a Consul+Vault+Nomad+Waypoint system, and start a Ceph cluster, all on the provided CWDC OpenStack resources. The resource needs (9 instances) rapidly grew to be too much for my system and Vagrant to handle, so I abandoned Vagrant in favour of testing on the production infrastructure, since it is vacant of projects anyways. I have written scripts to interact with Openstack to rebuild (reset) all of the CWDCs instances, which lets me rapidly destroy and deploy, which was the objective with Vagrant.

3.2 Security

I am currently adding ACLs, with Consul done, and Vault and Nomad in progress. Ceph I understand far less, I have deployed it using an automated tool, so I don't know how I should be fine tuning the configuration for speed and security. The auth proxy is nowhere near done. Waypoint, the one service that lacks ACLs, still needs to be split into zones.

The networking side of things are still wide open. Every node in the cluster has unrestricted access to all ports on other nodes. This greatly helped with setup, but is not a viable long term solution. I would like to experiment with Consul sidecars. These would allow me to lock down access between compute nodes to basically 0, and services running on two different hosts could communicate only if they explicitly both declared sidecars, special proxies that proxy traffic over one port from one host to another, for a specific service to access.

3.3 Redundancy/Recovery

I have occasionally restarted single hosts, however I would like to simulate a larger failure, and see how the cluster responds. This could include wiping Ceph storage devices, and shutting down multiple servers/clients, or even just killing Nomad job allocations.

4 Shortcomings

4.1 Waypoint

Waypoint is a very alpha piece of software. It does not support proper ACLs, just a binary access/no access based on unique user tokens. It is unable to revoke the sessions of users, the recommended way to do so is a clean reinstall and distribute new tokens to existing users. The Hashicorp team does have ACLs in the works, though I am unsure of how long until it is GA. Waypoint is the one piece of software I will have to split up into zones, and keep the higher priority projects away from the lower priority ones.

4.2 Redundancy

Since the entire service runs virtualized on openstack, a single openstack host failure could take down a critical amount of cluster nodes. Traefik is also not redundant, with a single instance being hosted on one of the nomad clients. This makes it easier as there is no coordination required when doing http challenges that letsencrypt requires before giving a certificate. It is possible to use consul to coordinate a HA pair of traefik instances, though I am not currently planning to do this.

4.3 Traefik TLS TCP Forwarding

Some of the hashicorp products, namely waypoint, use gRPC encrypted with a self signed TLS cert to communicate, with no option to disable it. Since Carleton ITS would likely need convincing to open up ports other than 80 and 443 in the firewall, we have to reverse proxy this TLS TCP gRPC traffic somehow. Since the waypoint user binary supports TLS, we can effectively use SNI to identify the host the TCP packet is heading for, and forward it to the waypoint server. Traefik, the main reverse proxy, unfortunately doesn't support re-encrypting raw tcp data going to the backend though, only ssl passthrough or termination. To have a by default (no need to install certs on the user's computer), trusted certificate presented for the gRPC

traffic, the only real option is to use traefik to terminate the TLS connection using a certificate automatically obtained from letsencrypt, proxy to a service like nginx in cleartext, and have that proxy to waypoint itself, using waypoint's self signed certificate. Hopefully this is addressed soon, as termination + re-encryption already exists for https traffic, just not TCP TLS.