Cloud native application to brute force MD5 passwords*

Chanthru Uthaya-Shankar[cu17780], Ainesh Sevellaraja[as17284], and Karthik Sridhar[ks17226]

University of Bristol

Abstract. Passwords provide a means to secure transactions over some potentially insecure medium and provide authenticity, giving a guarantee on a client brokering a transaction between some end point. MD5 is a widely used cryptographic hash function that produces a 128-bit hash value [1]. Using an MD5 hashing scheme presents many insecurities as anyone has the ability to uniquely pin down the plain text by only observing the encryption. This can be done through brute-force using a pre-image attack that is; for a hash function h, find an input x such that the image of x under h is the encryption. For any encryption using MD5 the search space for a potential x is constricted and constant. This search space can be partitioned and searched in parallel fashion utilising a collection of compute resources. These resources hash inputs and compare the output to the encryption. The attack is complete when an input is found that matches the encryption. Thus the authenticity of the client brokering the transaction is compromised. This paper briefly reviews an implementation of brute-force cracking an MD5 hash that leverages infrastructure made available by cloud providers such as Amazon Web Services (AWS).

Keywords: AWS - Amazon Web Services · k8s - Kubernetes · EC2 - Elastic Compute Cloud · S3 - Simple Storage Service · API - Application Program Interface

1 Outline of the Technologies incorporated

1.1 Amazon Web Services - AWS

AWS was used as a cloud provider to give the resources required to conduct our attack. AWS is an example of infrastructure as a service (**IaaS**) and provides many services as a part of its catalogue such as EC2. EC2 provides compute power through processors (CPU) and storage. In the case of this experiment, **m5.large** instance types running Ubuntu 20.0.4 were used due to their sufficient number of CPU's and available storage, making it ideal to run the implementation described in this paper.

* Supported by University of Bristol

1.2 Docker

Applications conducting the brute-force attack are containerised to provide a virtual environment for these applications to run in and are created using a built image. This encapsulation of applications inside containers provides a layer of isolation between other applications as well as the underlying OS kernel.[2] Docker was used as the container run-time engine. The portability of docker containers make it ideal for deploying the applications across the AWS infrastructure.[3]

1.3 Kubernetes - k8s

To orchestrate our application across multiple nodes, Kubernetes, an open source orchestration for deploying many micro-services was used.

Kubeadm was used to configure master and worker nodes to join the cluster through the token provided by the master node. A master node allows having orchestration components in a central control plane, with worker nodes making requests into the control plane via an API server. These orchestration components include:

- **Scheduler** - for scheduling the pods across nodes.
- **Etcd** - key value store for all cluster data in the case a pod goes down and needs to be replaced.
- **Control Manager** - for monitoring components.

These provide an excellent framework for maintaining and scaling the cloud application.

2 Architecture

All password requests are made to the front-end service, via the entry point node, which is the **master** of the k8s cluster.The front end service is exposed using a **NodePort**, allowing access to the application where the user can enter their password to be cracked. Applications communicate through **Rest APIs**, so HTTP requests can be made to the front end pods that contain information about the password. Rest APIs are used as the core functionality for communication between services in the cluster. Worker pods are responsible for the brute-force aspect of the system. Theses pods read in data that is published on a **queue system**. Finally, when a password has been cracked, the worker will write the password to an **S3 bucket**.

Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS service's IP to resolve DNS names. Each service exposes a list of endpoints which are the pods attached to that service. The service will **load balance** across all the pods attached to that service. All services of the system are part of the **default name**

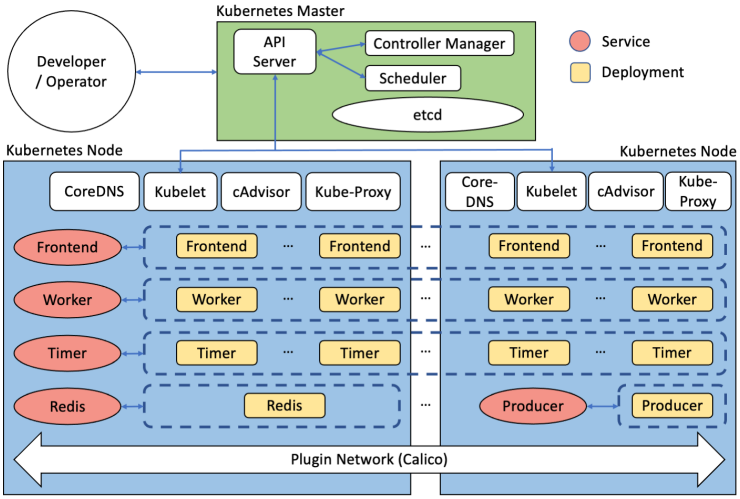

Fig. 1. Architecture Diagram

space. The service is reachable through `service-name.default.svc`, which resolves to the IP address of the service.

When down pods are replaced, the IP address of the pod also changes. The DNS assigned to the service provides a way of constantly routing packets to the pods behind the service even when a pod is replaced.

3 Key Parts of Implementation

3.1 Configuration

K8s requires that packets traversing a network bridge are processed for port-forwarding. Tunable parameters in the kernel bridge module must be set. Setting these tunable parameters allow bridged packets to traverse IP rule tables.

Docker and K8s components are installed. These include `kubeadm`, `kubectl` and `kubelet`. After this is done an image of an EC2 instance with the correct setup was saved to save time during the setup when creating new instances to join the cluster. This highlights another reason why AWS is ideal in this scenario. Kubeadm is used to initialize the k8s cluster. The kubeadm command `kubeadm init` bootstraps the k8s control plane. Kubeadm then installs a DNS server, **CoreDNS** which is essential for service discovery in the cluster. K8s isn't responsible for the network connectivity between pods. For this, a network plugin container network interface (CNI) must be added. Calico network is applied using a manifest from this[4] location. This Calico network will enable the CoreDNS pod to enter a ready state so that pods on the network can be discovered.

3.2 Cluster Management

A controller program that runs locally outside of the k8s cluster and monitors the cluster through kubectl commands by an SSH connection to the master node. Fabric allows orchestration of the setup of the cluster by making decisions on when to scale the amount of nodes to best match the workload imposed on the system. Boto3 was used for setting up EC2 instances. Nodes join a cluster using a token produced by the master on startup.

The controller then steps into a deployment phase where manifests are applied. These manifests consist of deployments and services. Deployment files contain the blueprints for a deployment. Metadata about the deployment is stored on the etcd, deployments are regularly checked along side the metadata stored in the etcd, so that the k8s controller and scheduler can make changes to the deployment to match the data on the etcd. When a pod goes down the scheduler will schedule new pods to match the replica size. This self-healing property of k8s is ideal as it provides a level of fault tolerance such that K8s will redeploy lost pods to match the deployment specification.

3.3 Cluster Components

Within these deployment files an image to be used in the container of the pod is provided. These are Docker images that are on a public docker registry, so can be pulled on deployment. The deployment files also defines port mapping for containers running inside of the pods. Environment variables are used to inject data inside the pod. The worker deployment pods use the AWS CLI credentials to publish results to an S3 bucket. Other deployment files include the front-end-deployment and the producer-deployment. These are NodeJS and Flask application respectively, the NodeJS application is responsible for taking in the user inputs and sending a request to the producer deployment to start publishing work to a Redis queue. A Redis deployment is used to setup our Redis store. All of these deployments are exposed through services. These services are defined in service manifests which use label selectors to group the pods that are part of the service.

The core of the work is done between the producer pods and worker pods. Producers split the search space into batches and publish these batches to be searched onto a Redis queue, worker pods take these batches and compute MD5 hashes of items defined by the batch data. Workers lease an item from the main queue by popping it and then push it onto a working queue, when work is completed the worker deletes the job from the work queue. The lease only allows the worker who picked up that job to work on it, when the lease expires the job can be leased to another worker if it isn't completed. This implementation allows a job to only be finished once and ensures that the job will be done to completion. A queue decouples the producers from the workers so that the publishers are not waiting for a response from the workers. This implementation works far better than a previous implementation tested where the producer sends data to be worked on through HTTP requests. This implementation hinders the scalability of the worker pods depending on the number of inputs.

4 Metrics

To scale the worker pods per the requirements, the horizontal pod autoscaler (HPA) provided by k8s was used. This automatically scales the pods in a deployment based on the CPU usage, which was checked through the **custom metrics support** (a component implemented as a k8s API). Based on the CPU limits and resources specified in the deployment file, the controller periodically adjusts the number of replicas in the deployment to reach the target. To get the resource metrics of each pod, the metrics server API was used. The metrics server provides the `/apis/metrics.k8s.io` path, which is launched as an api service. K8s provides a components deployment file[5] which allows us to deploy the metrics server as a service. The metrics server collects the pod wise metrics from the Summary API which is exposed by `kubelet` on each node.

Based on a target value, the controller calculates the usage value as a percentage of the equivalent resource request on the containers in each pod. The autoscaler controller operates on the ratio between the desired metric and current metric value which is calculated by:

```
desiredReplicas = ceil[currentReplicas * (currentMetricValue /
                      desiredMetricValue)]
```

as per the k8s documentation. The `currentMetricValue/desiredMetricValue` ratio is calculated for the pods that are fully ready only and disregards the non ready pods. K8s provides a simple command to create the horizontal pod autoscaler using the `kubectl autoscale` command.

4.1 Using the Metrics Server API for Scaling

The scaling monitor function is a unique way to scale the pods using the information gathered by the metrics server API. The monitor would periodically (every 1 minute) collect information and check whether to scale or not. The development process consisted of **three** main steps:

- **Information per pod name space and status** - The `kubectl get pods --all-namespaces` command gives information about each pod's corresponding namespace and status (running/pending/terminating etc). This was logged into `podstatus.txt`
- **Information per pod, status and its corresponding node** This was logged into `podloc.txt`.
- **Information of the CPU metrics per node** This was logged into cpumetric.txt

4.2 Scale and Terminate Function

The objective of this function was to identify:

- **The pending pods and their corresponding name spaces** - With the `podstatus.txt` file saved, a dictionary data structure was used to store the pod names and its namespaces as key value pairs. The pod statuses were checked and if it were "Pending", the count of the pending pods was incremented.
- **The nodes to terminate/scale down based on CPU metrics** - With the `cpumetrics.txt` file saved, the information was used to scale down the corresponding pods. The function keeps track of the nodes that went down due to unexpected reasons as well as the nodes eligible for termination (based on CPU usage). The CPU usage and IP address of the nodes were read and stored in a dictionary sorted by the CPU usage. A threshold of 50% was set to check whether a node should be terminated or not.
- **The pods to terminate/scale down from the shortlisted nodes from above** - The final part was to identify the pods deployed onto the nodes that have been terminated/shortlisted to scale down. K8s set the node IP address to *none*, if the node went down unexpectedly. This was useful in terms of ignoring the nodes that were still creating and pending.

A check was added to ensure that there must be at least two nodes below the CPU threshold, so that in the case of pod redeployment the pods will not overload on the active nodes. For each eligible node for termination, all the pods were checked and set to terminate except in the case of the Redis server.

4.3 Monitor mode

The prerequisites were used effectively inside a loop that monitors the cluster for potential scaling operations every minute. In terms of scaling down pods on already terminated/down nodes, the pods were forcefully terminated. On the other hand, if the average CPU usage was more than 80% and pending pods still existed, there was a need to scale up and create a new node. For scaling down, the nodes would be terminated based on the usage, if exists.

5 Methodology

Table 1 shows that for small sized inputs both the system with 8 nodes and 4 respond identically. This is due to the ratio between search space and batch sizes of jobs being very small. But as the number of letters grows the search space becomes exponentially larger causing more jobs to be backed up on the queue. By 7 letters the system with less worker nodes cannot consume and process data from the queue as quickly as the system with more workers, so results in a much larger time.

5.1 Scaling

Table 2 shows periodic metrics taken from the cluster when given a 7 letter input. The cluster is initialised with 3 worker nodes. By 120 seconds the HPA

Table 1. Time taken (seconds) to break the password from lengths 1-7

Nodes	L=1	L=2	L=3	L=4	L=5	L=6	L=7
Four	0.01	0.02	0.03	0.61	2.7	8.3	1860
Eight	0.01	0.01	0.02	0.63	2.8	4.1	285

scales pods to match the CPU usage target. This increase in pods then causes an increase in nodes, until the CPU usage stabilizes at 75%. Between 360 and 420 seconds the system finds the password and workers stop, causing a decrease in CPU usage.

Table 2. Metrics of a Scaling Cluster

Time (s)	60	120	180	240	300	360	420
Pods	20	27	32	39	45	45	45
Nodes	3	4	5	6	7	7	7
CPU %	100	96	98	99.6	100	75	63

5.2 Fault testing

Table 3 shows periodic metrics taken of a cluster when given a 7 letter input, the cluster is initialised with 3 nodes. The cluster starts to scale and at 300 seconds an error occurs and 2 worker nodes are lost from the cluster. This causes a drop in pods and drop in CPU usage from 96 to 63, as the cluster redeploys lost pods, eventually the CPU usage gets back up and the number of pods stabilizes.

Table 3. Metrics of fault testing the cluster

Time (s)	60	120	180	240	300	360	420	480	540
Pods	38	38	45	52	59	48	52	54	54
Nodes	4	4	5	6	7	5	6	7	7
CPU %	100	100	88.2	99.6	98.3	96	75	63	80

6 Discussion

Two main shortcomings were apparent when testing the fault tolerance of the system. That is the dependence on the master node and the Redis data store. In our architecture there is one control plane for the entire cluster, it is essential

for pods to use the control plane components to function. This can be mitigated by using a multi-master architecture where a load balancer is configured with endpoints that route to the kube API server in each control plane. This means if one master goes down its traffic would be routed to the other. It is worth noting that this was attempted in `fabric_scripts/multi-cluster.py`. This architecture includes only one Redis store in the from of a Redis pod, if a node goes down that the pod is on the whole system fails due to the producer and consumer dependencies on the queue. This can be mitigated with the use of Redis slave-replicas. The Redis queue being the only queue was the source of a bottle neck, in the process of decoupling the producers and consumers in a pub-sub system as the system scales the consumers also scales and end up idling due to a limit on the through put of a queue in which case the system scales down the idle workers. A possible solution to this is to use multiple queues where the workers get a choice on creation time to subscribe to a particular queue based on how busy it is. The implementation used in this paper only involves using lowercase letters, further research can be done into testing this system with longer inputs and a greater set of characters that include ASCII characters.

7 Conclusion

These results show the capabilities of using a micro-application container like architecture in k8s that can scale to meet demand as well as in the unlikely event of a node going down can withstand and recover. The self-healing properties of k8s cluster aids with the fault tolerance and keeps the service highly available, it is also flexible in deployment environments as demonstrated on AWS. This implementation involves a lot of things being built from the ground up including setting up, configuring, maintaining and scaling the entire cluster however there exists many available services for doing this. Amazon web services include EKS for managing and setting up the cluster and allows having a multi-master cluster. EC2 has auto scaling groups that will decide when to scale a cluster. These scaling algorithms are far more efficient than ones implemented in this paper.

References

1. https://en.wikipedia.org/wiki/MD5
2. Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71-79.
3. Vase, T. (2015). Advantages of Docker
4. https://docs.projectcalico.org/v3.14/manifests/calico.yaml
5. https://github.com/kubernetes-sigs/metrics-server/releases

A Appendix

Source Code Repository - https://github.com/ccdb-uob/CW20-04