Securing Your Collaborative Jupyter Notebooks in the Cloud using Container and Load Balancing Services

Haw-minn Lu‡*, Adrian Kwong‡

Abstract—Jupyter has become the go-to platform for developing data applications but data and security concerns, especially when dealing with healthcare, have become paramount for many institutions and applications dealing with sensitive information. How then can we continue to enjoy the data analysis and machine learning opportunities provided by Jupyter and the Python ecosystem while guaranteeing auditable compliance with security and privacy concerns? We will describe the architecture and implementation of a cloud based platform based on Jupyter that integrates with Amazon Web Services (AWS) and uses containerized services without exposing the platform to the vulnerabilities present in Kubernetes and JupyterHub. This architecture addresses the HIPAA requirements to ensure both security and privacy of data. The architecture uses an AWS service to provide JSON Web Tokens (JWT) for authentication as well as network control. Furthermore, our architecture enables secure collaboration and sharing of Jupyter notebooks. Even though our platform is focused on Jupyter notebooks and JupyterLab, it also supports R-Studio and bespoke applications that share the same authentication mechanisms. Further, the platform can be extended to other cloud services other than AWS.

Introduction

This paper focuses on secure implementation of Jupyter Notebooks and Jupyter Labs in a cloud based platform and more specifically on Amazon Web Services (AWS) though many architectures and methods described here are applicable to other cloud platforms.

Project Jupyter includes Jupyter Hub which provides a proxy and container management. In particular the Zero to Jupyter Hub with Kubernetes project, [Pro20] provides a framework to implement Jupyter Hub on a Kubernetes platform. The drawback is that Jupyter Hub is not equipped to easily secure. In addition, Kubernetes is notoriously difficult to secure and has many vulnerabilities that are not addressed by default.

The system described herein uses AWS's Elastic Container Service (ECS) for container management and AWS's application load balancer for authentication.

Described in this paper is an implementation which satisfies privacy and security concerns. However, the reader is advised to tailor the system to suit one's own needs.

Architecture

There are two distinct levels of architecture described. The cloud architecture comprises the various cloud services which is the lower layer of virtualization. The container architecture is the top layer virtualization built on top of the cloud architecture.

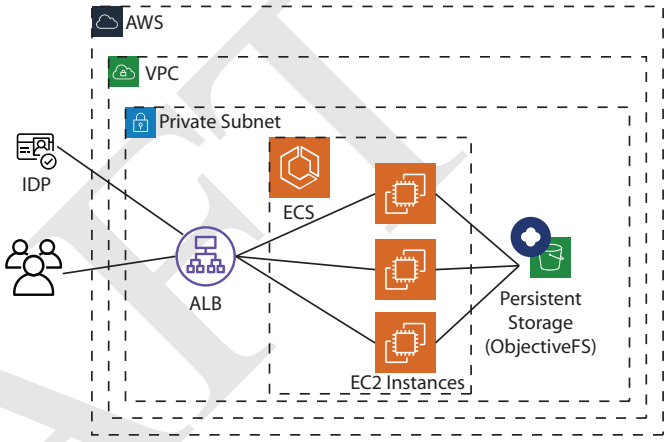

Fig. 1: *Cloud Architecture.*

Cloud Architecture

The basic cloud architecture is shown in Fig. 2. It comprises an identity provider (IDP) used to authenticate a user, an application load balancer (ALB) which can be configured to regulate user access through authentication, a collection of elastic cloud computer compute (EC2) instances that are used to instantiate the containers. Finally, ECS manages the containers deployed on the EC2 cluster.

Elastic Container Service: ECS is a container orchestration service. A container is instantiated as an ECS *task*. ECS provides a resource called a *task definition* that allow for the configuration of the container image, the environment variables, command override and container port.

Taking the most naive approach, ECS can be instructed to start a task based on a task definition. After the task has fully started, the host among the EC2 instances and the mapped port (the port on the EC2 node which is mapped to the container port) is known. At this point, one could write a monitoring function to detect when a task has started, retrieve the specific host and mapped port and create a listener rule for the application load balancer.

Instead of this cumbersome procedure, ECS provides another resource called a *service*. A service can manage many aspects of tasks within ECS including the number of tasks and a *target group* associated with the service. For our purposes managing the number means selecting a desired count of 1 or 0 depending on

* *Corresponding author: hlu@westhealth.org*
‡ *Gary and Mary West Health Institute*

whether the container is running or culled due to inactivity. A target group is a collection of ports of hosts, or serverless resources such as AWS Lambda function to which a listener rule can direct network traffic. In short by specifying a target group to a service, the host and mapped port are automatically assigned to the target group when a task has fully started.

Application Load Balancer: AWS's ALB can comprise multiple listeners to support multiple protocols. To maintain security, enforcement of HTTPS should be maintained either by not including a listener for HTTP or provide an HTTP listener that redirects all requests to HTTPS.

AWS's ALB, through a listener, is able to direct external HTTPS requests to various components. Based on listener rules, a request can be directed on the basis of both the hostname and the path. As an example, we use a path to specify a user and a service such as jupyter (for example, `domain.com/user_id/jupyter` or `domain.com/user_id/rstudio`), this allows us to give each user their own container.

Each listener rule maps a path, hostname or both to a particular target group. Since we uses an ECS service, we can assign a particular service to a target group. The service then manages which ports and EC2 instances are part of the target group.

While the ALB can enforce encryption from the end user to the ALB, the container application (e.g., jupyter) should also be configured to listen only for HTTPS. In this manner, the communication from the end user to the ALB is encrypted as is the communication from the ALB to the container application, ensuring end to end encryption.

Furthermore the application load balancer is also configured to perform authentication from an OpenID Connect (OIDC) compliant IDP. This eliminates the need for multiple messages to be passed when using either SAML or OAuth. Upon authentication, the ALB attaches three fields to the header of the http request `x-amzn-oidc-accesstoken`, `x-amzn-oidc-identity` and `x-amzn-oidc-data` which can be used by the end application to confirm the user's identity and validate the authentication. An example of this process as implemented in a jupyter notebook is described below.

For our IDP, we use Okta since it allows us to federate identity services to additional sign on services. This allows us to onboard collaborators and allow the collaborators to manage their users.

Shared Storage: In order to facilitate persistence across containers and also collaboration, ECS orchestrates containers on EC2 instances instead of AWS's Fargate product (Fargate facilitates containers in a serverless fashion but does not provide a host to mount an ObjectiveFS file system). Persistent storage can be mounted on the underlying EC2 instances. Individual containers can access the persistent storage by bind mounting the persistent storage. To meet security compliance of encryption at rest, the persistent storage should be encrypted. We elected to use the third party ObjectiveFS for cost reasons though native AWS resources such as elastic file system (EFS) can be used provided that both the file system and the network communications to the file system are encrypted. [Ser20c] ObjectiveFS is a secure file system backed by AWS simple storage service (S3). It should be noted to meet encryption in transit compliance requirements that any network attached storage must have network communications encrypted. For example, the base network file system (*nfs*) protocol is not.

As a specific example with jupyter notebooks we mount persistent storage as `/media/home/`. For a given user say `user_a` we bind mount `/home/jovyan` to `/media/home/user_a` so that while in the container the user sees `/home/jovyan` the home directory the users files are actually stored in the persistent storage in a `user_a` subdirectory. This configuration has two advantages. Only one persistent volume is needed to support all users' home directories minimizing costs and within the container all users see /home/jovyan thus eliminating the need to build a separate jupyter container image for each user.

With this configuration, multiple services can use the same home directory. For example, in our R Studio deployment `/home/rstudio` is also mapped to `/media/home/user_a`. Furthermore, we also can provide a persistent volume for shared directories. For example, for all users on `project_a` we bind mount `/home/jovyan/projects/project_a` to `/media/projects/project_a` where the persistent volume is mounted to `/media/projects`.

Resource Summary: To securely implement the above cloud architecture, each container instance for each user has a set of resources associated with it. First, a task definition is created for each user, this enables customized bind mounts as described above. Additionally, custom environment variables or task commands can also be supplied through the task definition. The task definition can also direct logging the the appropriate AWS CloudWatch stream.

Each user also has a ECS service, ALB listener rule and target group associated with it. This allows the seamless management of connecting a user to the desired container instance.

Finally each service has an AWS IAM role associated with it, this ensures the user has only the access rights to our AWS cloud that are need by the user. Beyond the rights to operate the container task, additional rights might include access to certain S3 storage or certain AWS Secrets Manager. As an example, we use the AWS Secrets Manager to manage user's credentials to various databases and public/private keys.

To simplify management of the per user resources, an AWS CloudFormation template is used to ensure consistency and uniformity among cloud resources whenever a new container instance/user combination is spun up. As an example, our CloudFormation template contains an IAM role, listener rule, target group, task definition, and an ECS service. Each template is then customized to spin up a CloudFormation stack for each user and application combination.

Container Architecture

The architecture in terms of container comprises a persistent hub container, an optional ephemeral provisioner container, and an assortment of semi-persistent application containers such as jupyter notebook. In an alternative deployment, AWS Lambda functions can be functionally substituted for the hub container, but for the sake of simplicity only the container version of the hub is described.

The application containers are described as semi-persistent as they can be started on demand and culled when one or more inactivity criteria has been reached. This can be achieved by updating the associated service to have a desired count of `1` to start or a desired count of `0` to cull.

We adopted a url path routing convention to access each application such as `domain.com/user_id/application`

Container Management: The heart of the system is the hub container. To facilitate ALB authentication, two listener rules are provided. One rule allows anyone to connect, so that the hub

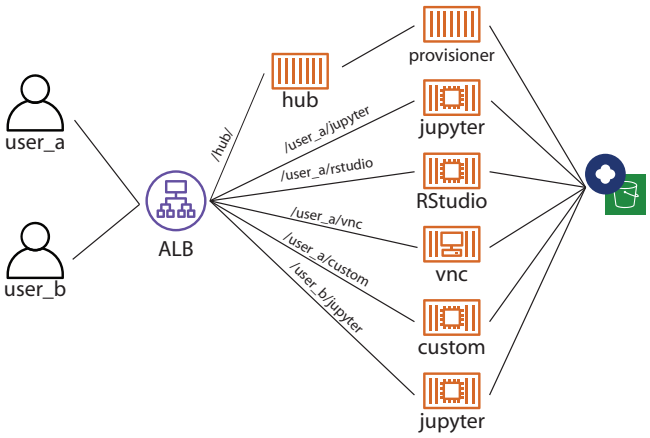

Fig. 2: *Cloud Architecture.*

can present a login page (with single sign on and IDP this looks like a single login button). The login action redirects the browser to a url which forces authentication via the ALB. Though this step is not necessary, it provides a cue that makes for a smoother user experience.

Since the hub container may be given privileges to set IAM roles for the application services, the role under which the hub service runs can have a boundary policy attached to it [Ser20d]. This ensures that any role created by the hub service is constrained to include the boundary policy. This prevents the hub from being able to create an arbitrary role should the container become compromised.

The provisioner container is an ephemeral task which is run with the persistent storage mounted. The provisioner can create a home directory for a user the first time the user logs in and provision the directory with any necessary files. While the functionality of the provisioner container could be incorporated in the hub container. Separation allows the provisioner to run with minimal cloud privileges (IAM role) and allows the hub to have no access to the shared home directory, so in the event the hub container is compromised the user's file system is not exposed. Also, with separation the hub does not have to have access to the file system so it can be refactored and deployed as a Lambda function. Furthermore the provisioner container runs very briefly further limiting the vulnerability window.

Once authenticated, the user can elect to connect to an application container. This can occur under three circumstances: the user's application container is still running, the user's application container has been culled, or the user has never started the application before. If the container is still running, the user is immediately redirected to the container. If the container has been culled, the service is updated to a desired count of `1`. If the application has never been started by the user, resources to spin up the service are created such as by creating a CloudFormation stack.

Additionally, an option to "decommission" an application can be presented where the CloudFormation stack can be deleted.

Culling: The best practice for culling an application is to have the application upon exiting, set the desired count to `0` of its corresponding service.

For the example of jupyter, the start up scripts for both jupyter notebook and jupyter lab contains the following snippet with `main` imported from different places:

```
if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$',
                         '', sys.argv[0])
    sys.exit(main())
```

Rather than just exiting after `main` completes, a modified start up script updates the desired count of the corresponding service to `0`. Since `boto3` essentially wraps API calls to AWS, a delay before termination is needed to ensure the update API call is received before terminating the task. Failure to change the desired count will only result in the service restarting the container upon termination.

```
if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$',
                         '', sys.argv[0])
    main()
    session = boto3.Session()
    ecs = session.client("ecs", region_name)
    ecs.update_service(cluster=cluster_name,
                       service=service_name,
                       desiredCount=0)
    # Sleep for 2 minutes give service time to update
    time.sleep(120)
```

Code to retrieve the `region_name`, `cluster_name`, and `service_name`, are omitted for clarity, but they can be retrieved from environment variables (set in task definition), passed via `sys.argv` or even by calls to `boto3`. Though the first two options are simpler.

The above modification to the start up scripts ensures that when jupyter exits the task count is zero. However, in order for this to be meaningful culling parameters in the jupyter configuration such as `c.NotebookApp.shutdown_no_activity_timeout` `c.MappingKernelManager.cull_connected`, `c.MappingKernelManager.cull_idle_timeout` and `c.MappingKernelManager.cull_interval`, as well as setting a shell timeout (e.g., `TMOUT` environment variable are set) in the event a terminal is open.

Authentication and Applications

As mentioned above, the bulk of the authentication is performed by the ALB. However, it is important for the individual application to validate a request forwarded by the ALB, for two reasons. Validation prevents potential security vulnerablities due to a misconfiguration in the system or exposes security vulnerabilities during the initial system debugging. Additionally, validation ensures that the identity of the user is what is expected. The ALB ensures that the user has validly authenticated, but it is up to the application to ensure that the correct user has connected.

Validation is achieved through the JWT token presented in the `x-amzn-oidc-data` header by the ALB. These JWT tokens are signed by a public key retrievable from AWS insuring that only the ALB could have signed them. Within the JWT token, the `kid` field represents the *key ID* for the public key. To validate, the key ID should be extracted and corresponding public key should be retrieved from AWS. With the public key, the JWT token can then be validated. We use the `python-jose` module available on PyPi. The `sub` field in the JWT token is the same as the OIDC ID which is also presented in the `x-amzn-oidc-identity` field. The application should then verify this is OIDC ID associated with the expected user.

To deploy an application securely in our infrastructure, in addition to validating the authentication, the application container

should meet four more requirements. It should have a configurable base url as the ALB will forward requests to the application with the base url prefix. It should communicate to the ALB over HTTPS to ensure end to end encryption. It should provide a url to respond to pings sent by the ALB for health checks. It should validate that the mounted home container belongs to the user.

The solution to the last requirement is for our provisioner to write an `.id` file in the user's home directory containing the user's ID. This file is written by `root` and is only readable. The application upon startup or authenticaation can verify that the user has the correct home directory mounted. This requirement is a safeguard against misconfiguration and can be omitted if one is confident that the system is not misconfigured.

Jupyter

Unfortunately, unlike JupyterHub, jupyter notebook/lab do not come with a pluggable authentication module. In order to implement validation, the source file `login.py` must be modified. This file is usually located in the `notebook/auth/` directory in your `site-packages` or `dist-packages` directory. Since Jupyter notebook and JupyterLab are not truly separate applications (in fact they are interchangeable using the path `/tree` or `/lab`), the same `login.py` file facilitates authentication for both. If you build using a standard docker image such as `jupyter/base-notebook` or any of its derivative notebooks, this directory would be `/opt/conda/lib/python3.x/site-packages` directory. Please note that the specific python version may vary dependent on which version of the docker container is used and whether subsequent additional install modules might force a rollback of python versions.

The specific modification to the `login.py` file involves replacing two methods, the `get` method and the `get_user_token` class method of the `LoginHandler` class.

Unaltered, the method `get` determines whether the `current_user` is set indicating the user has been logged in. If not authenticated, the function presents a login page. Our modification simply adds an additional check that if `current_user` is not set, we validate the JWT token in header to determine additionally whether the user is authenticated. It should also be noted that the function is also decorated as a coroutine to make the function asynchronous as the verification may require network access to retrieve a public key.

```
@tornado.gen.coroutine
def get(self):
    authenticated = False
    if self.current_user:
        authenticated = True
    else:
        if self.verify_jwt():
            authenticated=True
    if authenticated:
        next_url = self.get_argument('next',
            default=self.base_url)
        self._redirect_safe(next_url)
    else:
        self._render()
```

The other method to be replaced is the `get_user_token`. Unaltered, the method returns the authorization token used as part of a notebook/lab minimal authentication scheme. This token is normally supplied as a query string in the URL or through the login page. We bypass this mechanism altogether. Instead, we examine the request header for a JWT token supplied by AWS and validate it. If it is successful we provide a token. As far as the rest of the notebook code the value of the token is not used so we supply a random string. Our version of `get_user_token` uses a local cache to store retrieved public keys and previously the previously decoded user ID.

```
@classmethod
def get_user_token(cls, handler):
    """Identify the user based on
       Authorization header

    Returns:
    - uuid if authenticated
    - None if not
    """

    authenticated = False
    if cls.verify_oidc(handler):
        authenticated = True
    else:
        oidc_jwt = handler.request.headers\
            .get('x-amzn-oidc-data')
        if oidc_jwt:
            try:
                header = jwt.get_unverified_headers( \
                    oidc_jwt)
            except JOSEError:
                return None
            kid = header.get('kid')
            if kid and kid == user_cache.get('kid') \
                and user_cache.get('pk'):
                try:
                    token = jwt.decode(oidc_jwt,
                                       user_cache['pk'])
                except JOSEError:
                    return None
                oidc_id = handler.request.headers\
                    .get('x-amzn-oidc-identity')
                if token['sub'] == oidc_id:
                    authenticated = True
                    user_cache['jwt'] = oidc_jwt
                    user_cache['user_id'] = oidc_id
    if authenticated:
        return uuid.uuid4().hex
    else:
        return None
```

In addition to the two modified methods, we supply two helper methods `verify_jwt` for `get` and `verify_oidc` for `get_user_token`. They perform the token validation and cache management. Additional code which can read identifiers in persistent volumes and verify they match the user who is authenticated can also be added to ensure two authenticated users don't have access to the other's containers.

```
def verify_jwt(self):
    global user_cache
    oidc_id = self.request.headers\
                .get('x-amzn-oidc-identity')
    oidc_jwt = self.request.headers\
                .get('x-amzn-oidc-data')

    if not oidc_jwt:
        self.log.warning("No JWT Token in Header")
        return False

    if (user_cache.get('user_id') == oidc_id and \
        user_cache.get('jwt') == oidc_jwt):
        return True

    try:
        header = jwt.get_unverified_headers(oidc_jwt)
    except JOSEError as e:
        self.log.error("JWT failed to decode: {}"\
                    .format(e))
        return False
```

```
    kid = header.get('kid')
    if not kid:
        self.log.error("No Key ID in JWT token")
        return False

    if kid != user_cache.get('kid'):
        if 'pk' in user_cache:
            del user_cache['pk']

    if not 'pk' in user_cache:
        try:
            r = requests.get(PK_SERVER + kid)
            # TODO treat return code
            user_cache['pk'] = r.text
            user_cache['kid'] = kid
        except requests.RequestException as e:
            self.log.error("Requests Error: {}"\
                           .format(e))
            return False

    try:
        token = jwt.decode(oidc_jwt,
                           user_cache['pk'])
    except JOSEError as e:
        self.log.info("JWT failed to validate: {}"\
                      .format(e))
        return False

    if token['sub'] != oidc_id:
        self.log.error("User ID in token doesn't "
                       "match user ID in header")
        return False

    user_cache['user_id'] = oidc_id
    user_cache['jwt'] = oidc_jwt

@classmethod
def verify_oidc(cls, handler):
    global user_cache
    oidc_id = handler.request.headers\
                .get('x-amzn-oidc-identity')
    oidc_jwt = handler.request.headers\
                .get('x-amzn-oidc-data')

    if not oidc_id or not oidc_jwt:
        return False
    if oidc_id != user_cache.get('user_id'):
        return False
    if oidc_jwt != user_cache.get('jwt'):
        return False
    try:
        header = jwt.get_unverified_headers(oidc_jwt)
    except JOSEError:
        return False
    kid = header.get('kid')
    if kid != user_cache.get('kid'):
        return False

    return True
```

To meet the other requirements for jupyter, the `base_url` configuration needs to be set to ensure that the route is properly interpreted. Furthermore, we use this `base_url` as the health check url which responds with a `302` code. A self-signed certificate is automatically generated when the container starts and that certificate is then used to configure jupyter to run over HTTPS.

RStudio

Our implementation of RStudio Server on the same cloud platform is non-invasive to the code base, but more complicated architecturally. Since RStudio does not have a way to set the base URL of the application, a proxy is required to rewrite the HTTPS request paths. We use an `nginx` proxy to rewrite requests to RStudio Server using the `proxy_redirect` directive.

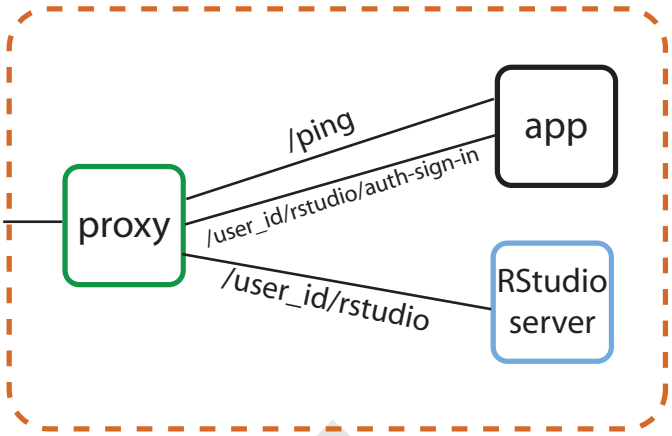

Fig. 3: *Inside the RStudio Container*

Figure 3 shows the application structure within the RStudio container. A proxy communicates with the ALB and routes some requests to a custom app used for authentication and handling the health checks and others to the RStudio server. Since communications between the proxy, app and RStudio server are all within the container and not exposed, they do not require encryption to satisfy compliance. A self-signed certificate is created upon container startup that enables `nginx` to communicate over HTTPS to the ALB.

For authentication, RStudio Server maintains authentication session information in a cookie. So with `nginx` we capture, the `auth-sign-in` URL and redirect it to an lightweight webapp whose sole function is to authenticate the user, set the cookie and redirect the browser to RStudio Server. Since the app is necessary in this configuration, we also configure the app to respond to a `/ping` request issued by the ALB target group's health check.

The authentication code is nearly identical to the `verify_jwt` function written above for jupyter. The cookie consists of three pieces, a user ID (which we retain as the default `rstudio` as we retained `jovyan` for the jupyter notebook, to prevent the need to build a separate docker image for each user), the expiry and an HMAC 256 signature, signed with a secret typically stored at `/var/lib/rstudio-server/secure-cookie-key` inside the container. The following snippet of code implements this.

```
from urllib.parse import quote
from Crypto.Hash import HMAC
from Crypto.Hash import SHA256
import base64
import datetime

utc = datetime.datetime.utcnow()
expiry = utc + datetime.timedelta(days)
now = expiry.strftime('%a, %d %b %Y %H:%M:%S GMT')
dig = base64.b64encode( \
        HMAC.new(secret,
                 "{0}{1}".format(username, now),
                 digestmod=SHA256).digest())

cookie = quote("{0}|{1}|{2}".format(username,
                                    now,
                                    dig.decode()),
               '|')
response.set_cookie('user-id', cookie)
```

The `days` is the number of days til the cookie expires, and `username` is the user name (i.e. `rstudio`). In the above snippet, the cookie is attacked to a Flask response.

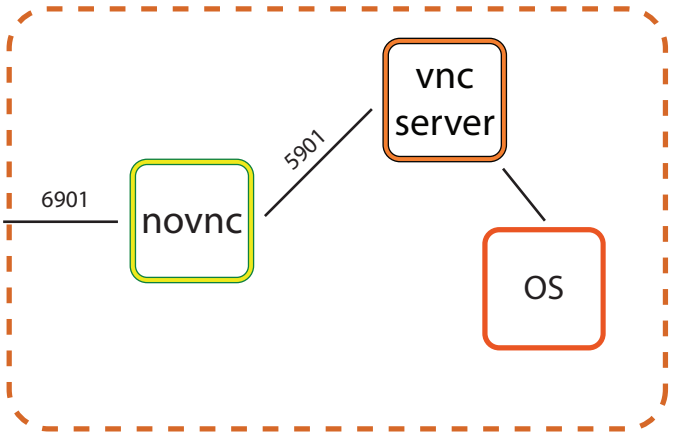

Fig. 4: *Inside a VNC Container*

Virtual Network Computing (VNC) Containers

There are many desktop apps for Linux which may also be useful to deploy via a web application on a cloud cluster such as presented here. The following implementation allows the deployment of such applications such as Orange and Falcon through the use of a web VNC client to a VNC server running in a container.

This is based on the Docker Headless VNC Container project [Con19] as a blueprint using the `xfce4` window manager. Since it appears that the project has been inactive for over a year we adopt its `Dockerfile` as a starting point but do not use the docker images as a building block.

Figure 4 shows the application structure within a headless VNC container. NoVNC [noV20] is used as a web to vnc proxy which connects via VNC to a local vnc server which in accordance to the Docker Headless VNC Container project is tigerVNC [Tig20]. Through the VNC server graphically oriented operating system commands and applications can be executed. In our container tigerVNC is unchanged and is installed just as it is in the headerless project's `Dockerfile`. The noVNC project comprises a `novnc` and `websockify` component. No changes were made to the `novnc` component except to alter the parameters use to start `websockify`. Therefore the focus of the customization is on the `websockify` component.

Fortunately, `websockify` permits authentication plugins. The plugin is a simple class with an `authenticate` method which accepts the `headers`, `target_host` and `target_port` as parameters. Upon success it returns and on failure it raises an `AuthenticationError` exception. Since the body of the code is essentially the same as the *verify_jwt* method descirbed for jupyter, the code is not repeated here.

It should be noted that in the container by default the VNC server listens on port 5901 and the novnc client listens on port 6901. It is recommended that only port 6901 be exposed so that only the novnc client can directly communicate with the VNC server as the VNC password in this environment is not well protected. By only exposing port 6901, knowledge of the VNC password can not be exploited to bypass the authentication.

Furthermore, the web server within the `websockify` project is located in `websockifyserver.py` and is based on `SimpleHTTPServer`. It may be desirable to create a custom handler or custom `do_GET` method to handle issues such as providing a base URL, health check URL for the ALB's target group, or to implement templating if desired. A self-signed certificate is generated in a *launch.sh* as *self.pem* which the webserver will automatically detect and run using HTTPS.

Once this base container image is build with those customizations. Applications such as Orange or Falcon can be added, thus not limiting the cloud system to web applications.

Custom Applications

In developing your own bespoke applications, a layer of authentication can be employed. In consideration of developing or adapting your own application, you should provide an unauthenticated URL for the ALB's health check and be equipped to configure the base URL. Authentication can be easily plugged into most web server frameworks.

As a simple example, using flask authentication can be incorporated into a custom `login_required` decorator, so that for any protected URL the request is authenticated before being processed. Once again the decorator could be implemented with code similar to that of *jwt_verify* described above.

Security and Compliance

In our cloud architecture, the bulk of the security and compliance is built into the EC2 instances serving as nodes behind the ALB and built into features of the ALB. By keeping most of the security external to the containers, container images need less customization for security purposes making it easier to support a wide variety of container images and container apps.

The preferred method to implement security, compliance, and even maintenance services on an EC2 instance is to install the appropriate software in an Amazon machine image (AMI). By building a customized AMI based off an optimized Amazon ECS reference AMI [Ser20b] but including the desired additional services installed, an fully equipped EC2 instance can be spun up quickly and features such as autoscaling can easily be applied.

Specifics to security and compliance implementations are described in the following subsections including encryption at rest, access controls, auditing and other agents.

Encryption at Rest

As previoiusly mentioned, persistent storage and associated file system protocol are encrypted give both encryption in transit and encryption at rest for the persistent storage. However, it is also important that the base file system of the EC2 instances are also encrypted to fully ensure encryption at rest. There are two important aspects of ensuring encryption at rest for the base file system. First the attached file system such as elastic block storage (EBS) must be encrypted. This is accomplished by selecting encryption when creating the EC2 instance or within a launch configuration. Fortunately, AWS now offers an account-level option where EBS volumes are encrypted by default for any EBS volumes created in that given account. We highly recommend this option as it will mitigate the chances of misconfiguration. »»»> 0d077eb728eb153cc464729ef98c18449e634364

Furthermore, the AMI used to create EC2 instances must also be encrypted. A common technique for doing so is to build an machine snapshot will all the agents and services desired then encrypt the snapshot. Regardless for what techinque is used. the AMI's should be encrypted to satisfy any requirements for encryption at rest.

Access Control

Another security concern is controlling the internet access from the container. The reason is two fold. First, controlling access allows us to prevent users from within a container from accessing potentially malicious websites. Second, should a container become compromised we want to mitigate the compromised container's ability to escalate privileges or pivot to other services within the organization. While AWS through the use of security groups and access control lists provide a coarse ability to regulate what destinations are accessible, we favor more fine grain control.

There are two aspects of this finer grain control, first we use an on-host firewall to control outbound access from the hosted containers. Second we funnel all traffic from each container to a proxy.

For the firewall, we use `iptables` using the following commands:

```
iptables --insert DOCKER-USER --in-interface docker0 \
   -o eth0 -j DROP

iptables --insert DOCKER-USER \
   --destination 169.254.169.254 --jump REJECT \
   --reject-with icmp-port-unreachable

iptables -t nat -A PREROUTING -i docker0 \
   -d 172.17.0.1 -p tcp --dport 8888 -j RETURN
iptables -t nat -A PREROUTING -i docker0 \
   -d 172.17.0.1 -p tcp -j DNAT --to-destination :2
```

The first command blocks all internet traffic coming from the *docker0* interface (where the containers must route through) to the *eth0* interface which is the external interface. The second command (see [Cos18]) blocks access to the node specific metadata service, which typical contains information about the EC2 instance and credentials for that instance.d. Blocking this prevents a compromised container from accessing the metadata about the EC2 instances blocking a potential escalation in privliges to that of the EC2 node. The third and fourth commands allows the container access to the EC2 instance (which in the docker world is IP address `172.17.0.1`) only on port 8888, where the proxy is configured to listen. All other access is routed to port 2 which has no active listeners.

On the container side, the environment variables `http_proxy` and `https_proxy` must be set to forward all http and https request to the EC2 instance at port 8888. In addition the `no_proxy` environment variable should be set to allow some traffic not to be forced into the proxy. Of course, `localhost` (and corresponding IP address `127.0.0.1`) do not require proxy as the traffic doesn't leave the container. In addition, the metadata IP address `169.254.169.254` should be allowed out so that the `iptables` rule regarding the metadata traffic can be enforced. Finally, the IP address `169.254.169.2` is used by the ECS agent.

Two methods can be used to address the environment variables. Either we can add the environment variables to the task definition when an application service created or it can defined in the container's *Dockerfile* with the following lines:

```
ENV http_proxy=http://172.17.0.1:8888/
ENV https_proxy=http://172.17.0.1:8888/
ENV no_proxy=localhost,127.0.0.1,\
169.254.169.254,169.254.170.2
```

Because of the `iptables` rules a misconfiguration that fails to set the proper environment variables results in loss of access and not a vulnerability.

The proxy can then determine whether to route the connection request directly externally or through an external outbound gateway which could include a company firewall so that broad based policies could be applied. For the proxy we selected `tinyproxy` because it is lightweight and allows gateway credentials to be embedded in the proxy configuration pushing the burden of gateway credentials to the proxy and not the container or application of the container.

Auditing

Beyond security reasons, many regulations such as HIPAA require auditing for compliance. Our approach is two fold. We use the ALB logging capabilities to track access to application containers and authentication. We use a logging agent to track potential privlege escalation or other security concerns on the underlying EC2 host.

The ALB provides logging [Ser20a] which will log all access to the application containers to an S3 bucket. Because in our architecture all authentication is performed using the ALB all authentication attempts both successful and more importantly failures are also logged to the bucket. Many third party log management tools are configurable to digest logs stored in this manner including Loggly, Splunk, Sumo Logic.

Another good practice is to set the target S3 bucket in a separate AWS account and only grant privleges to the logging account to write to the bucket but not delete. This ensure that even if a container or the EC2 instance is compromised, the logs can not be tampered with.

To supplement the auditing and monitoring capability one or more logging agents are installed on the EC2 instance. Essentially, this agent transmits logs of interest such as the system log `syslog` to an external log management system. Through this mechanism behaviours such as privlege escalation (e.g. `sudo`) are tracked. We use both the native AWS logging agent and a third party logging agent.

With both mechanisms in place, the preferred log management system can be configured to provide alarms when severe incidents occurs and generate reports of incidents as may be required by compliance requirements.

Other Useful Agents

Building a custom AMI image to spin up an EC2 instance to support our ECS cluster affords the opportunity to install additional agents to meet security, compliance and maintenance needs. Our best practices is to the include the following additional agents in the AMI. Some of agents are provided by AWS while some are third party.

ECS Agent: The AWS ECS agent is required in order for the EC2 instance to serve ECS containers. However, periodically updating the ECS agent is important in that potential vulnerabilites may be fixed and newer agents offer more features to aid in maintenance. Furthermore, proper configuration of features can aid in security as well. For example, the ECS agent can be configure so that the maximum lifetime of an EC2 instance is set. This is particularly useful if the AMIs for the EC2 instances are constantly being updated with security patches etc. The limited lifetime guarantees that the EC2 instances running will not be based on an AMI that is too out of date.

Systems Manager Agent: Another useful AWS Agent that can be employed is the AWS Systems Manager Agent (SSM) [Ser20e]. The SSM agent allows the "Systems Manager to update,

manage and configure" the EC2 instances. This agent makes it easier to maintain EC2 instances in a centralized manner. Once again keeping an EC2 instance up to date helps reduce vulnerabilites on the node.

Anti-virus: An antivirus or antimalware agent is also recommended. The antivirus should be one that is container aware and that the container awareness feature should be active. This would facilitate pinpointing the specific container that may be compromised. Container systems such as docker are not complete virtualizations. Processes that run in a container run as processes in the native host, as such an antivirus agent inside can monitor processes that occur "inside a container". Container aware antivirus agents makes mitigation in a container environment easier. In our particular configuration, we use Sophos as the antivirus though you may have your own preferences.

Intrusion Detection: Another useful agent to be deployed on the EC2 instance is an intrusion detection agent. Like the antivirus agent, an intrusion detection agent that has container awareness capabilities is desireable and should have the capability activated. The intrusion detection agent looks for activities that are anomolous and when high risk activity is detected, it will gather as much information around the incident as it can. We use ThreatStack for our intrusion detection.

Conclusion

Presented here is a secure, collaborative infrastructure for deploying a cloud computation resources. The primary purpose of our infrastructure is to provide jupyter in this environment though due to the preference of some of our users RStudio and other tools are include. Our data science and infrastructure team is small so building a compliant infrastructure that requires little maintenance is paramount. Equally important is to safeguard against opening vulnerabilities due to misconfigurations. By following the suggestions presented here, misconfigurations err on the side of loss of functionality rather than introducing vulnerabilites.

The architecture presented here was successful in a recently performed penetration test. While the recommendations and architecture shown here rely heavily on AWS resources. No doubt elements and counterparts can be found in other cloud services such as Google Cloud and Microsoft Azure.

Snipets of code, Dockerfile, commands and other resources presented here and the corresponding poster are available at West Health's github repository at https://github.com/Westhealth/scipy2020/cloud_infrastructure.

References

[Con19] ConSol Misc GmbH. Docker container images with "headless" vnc session, 2019. URL: https://github.com/ConSol/docker-headless-vnc-container.

[Cos18] Ciro S. Costa. Blocking ec2 metadata service from docker containers in aws, Aug 2018. URL: https://ops.tips/blog/blocking-docker-containers-from-ec2-metadata/.

[noV20] noVNC. novnc, 2020. URL: https://novnc.com/info.html.

[Pro20] Project Jupyter. Zero to jupyterhub with kubernetes, 2020. URL: https://zero-to-jupyterhub.readthedocs.io/en/latest/.

[Ser20a] Amazon Web Services. Access logs for your application load balancer, 2020. URL: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-access-logs.html.

[Ser20b] Amazon Web Services. Amazon ecs-optimized amis, 2020. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html.

[Ser20c] Amazon Web Services. Data encryption in efs, 2020. URL: https://docs.aws.amazon.com/efs/latest/ug/encryption.html.

[Ser20d] Amazon Web Services. Permissions boundaries for iam entities, 2020. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_boundaries.html.

[Ser20e] Amazon Web Services. Working with ssm agent, 2020. URL: https://docs.aws.amazon.com/systems-manager/latest/userguide/ssm-agent.html.

[Tig20] TigerVNC. Tigervnc, 2020. URL: https://tigervnc.org/.