CAC in the Cloud

Jim Ladd

July 16, 2021

Contents

Introduction

SOFWERX is a non-profit organization with a Partner Intermediary Agreement (PIA) with the Special Operations Command Center (SOCOM). We address Warfighter problems through collaboration, innovation, events, and rapid prototyping. While SOFWERX deals only with non-classified data, technologies, and processes, there is some online information that needs to be restricted from the general public. This restriction ensures the privacy of intellectual property of vendors from their competitors.

The web development team was tasked with integrating the Department of Defense's Common Access Card (CAC) into our web security solution. The initial version relied on a reverse proxy server using AWS Elastic Container Service (ECS), Docker, and NGINX. After an eighteen month lifecycle, a new version was deployed using the same technology stack but this time as a micro-service. The latest version is a micro-service but uses AWS serverless technologies including Lambda, API Gateway, and S3. All three solutions relied on GitLab and its CI/CD pipeline service. This paper presents the three solutions that integrated the Common Access Card with cloud technologies.

Background

The Common Access Card (CAC) is a "smart" card and provides standard identification for active and reserve military personnel, Department of Defense civilian employees, and eligible contractors. The CAC is close in size to a credit card and enables physical access to controlled facilities. It also provides access to DoD-related computer network and systems [1].

The overall process of using the CAC and the DoD certificates is straightforward.

1. The user downloads the DoD certificates and installs them in the browser on a local computer.

2. The user inserts the CAC into a compatible reader.

3. The user navigates to a URL that requires a CAC.

4. The browser prompts the user to select one of the installed certificates.

5. The browser prompts the user to enter the PIN for the CAC.

6. The browser and the server at the URL will continue with the TLS handshake. During this process, the server will validate the CAC's private key with the installed public certificate.

7. Once validated the server will continue with the handshake process and eventually execution control is handed over to the next step in the server's process.

8. If the handshake fails, the server will respond either a network connection reset error or with a HTTP status code of 403.

The DoD public certificates may be downloaded from the DoD CYBER EXCHANGE website [2]. The Document Library section of the website contains a set of PKI CA Certificate Bundles. To find this set, navigate to the page and use the keyword "bundle". The latest version at the time of this writing is "PKI CA Certificate Bundles: PKCS#7 for DoD PKI Only - Version 5.8". A zip file can be downloaded from this page.

SOFWERX's use of the CAC originated from a non-traditional requirement. While our online information is not classified but we wanted to limit the access to members of the DoD and other government agencies and do so in an easy and low maintenance fashion. The motivation was to ensure a vendor's intellectual property was not available to competitors and not to burden the intended users with individual login credentials.

One of the important features of the DoD's CAC technology is that a unique number is assigned to the user during the CAC registration process. This number will not change during the lifecycle of the user's profile. The user may change his/her first or last name but this id will not change. SOFWERX wanted to validate that a user was a CAC holder only, we do not track or verify the individual information associated with the CAC. If our requirement changes in the future, this unique ID would be very useful.

SOFWERX deployed the first version of the CAC security solution in October of 2019. This design used a reverse proxy server that relied on the CAC technology to validate every HTTPS request. In June, 2021 we switched to a micro-service design that simply returned a status code of 200 if the CAC was valid. A few weeks later, the micro-service was migrated to a serverless stack on AWS. The remainder of this document presents each of the three architectures.

CAC Reverse Proxy Server

The initial design of the solution was based on a reverse proxy server concept. A reverse proxy server typically sits behind the firewall in a private network and directs client requests to the appropriate backend server. A reverse proxy provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers [3].

This approach configures the reverse proxy server to require a client certificate that is compatible with the DoD public certificates. The HTTPS requests for the "public" pages are routed directly to the web server. The requests for the "CAC protected" pages are directed to the reverse proxy server which validates the CAC of the user. Once approved, the requests are forwarded to the web server.

The design of the reverse proxy server is based on the work of Michael Pyne who authored the *mpyne-navy/nginx-cac* GitHub repository [4]. This code configures a NGINX web server with the DoD certificates in a Docker image. The SOFWERX approach configured the NGINX as a reverse proxy server and then hosted the Docker image using the Elastic Container Service (ECS) in Amazon Web Service (AWS). A diagram of the architecture is shown below:
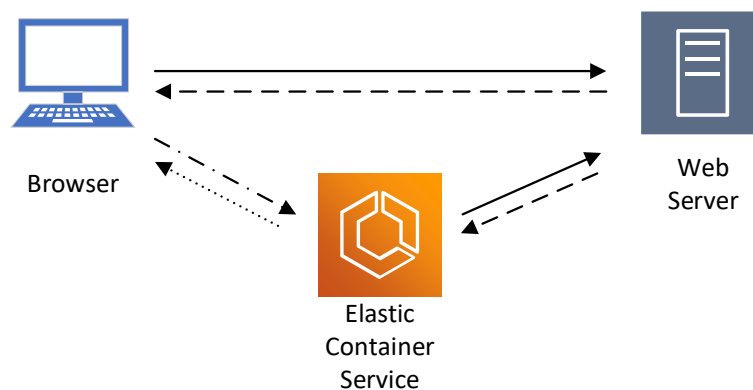

Browser

Elastic
Container
Service

Web
Server

Figure 1 – Reverse proxy server

The configuration file for NGINX (default.conf) can be found in Appendix A. The Dockerfile is presented in Appendix B.

Options available with the NGINX configuration allows access to 1) the user's unique CAC identification number and 2) status of the verification. The *$ssl_client_s_dn* variable generates the following value:

```
CN=DOE.JOHN.PAUL.1603885506,OU=CONTRACTOR,OU=PKI,OU=DoD,O=U.S. Government,C=US
```

Similarly, the *$ssl_client_verify* variable generates the following value:

SUCCESS

These values can be inserted in headers or used in logic within the configuration file.

GitLab was used for the source code repository service and its CI/CD pipeline service was configured to build and then copy the Docker image to the AWS Elastic Container Registry (ECR).   An AWS CodePipeline was created that deploys new images to the Amazon Elastic Container Service (ECS). The ECS is a fully managed container orchestration service that eases the deployment, management, and scaling of containerized applications [5]. A diagram of the process is shown below:
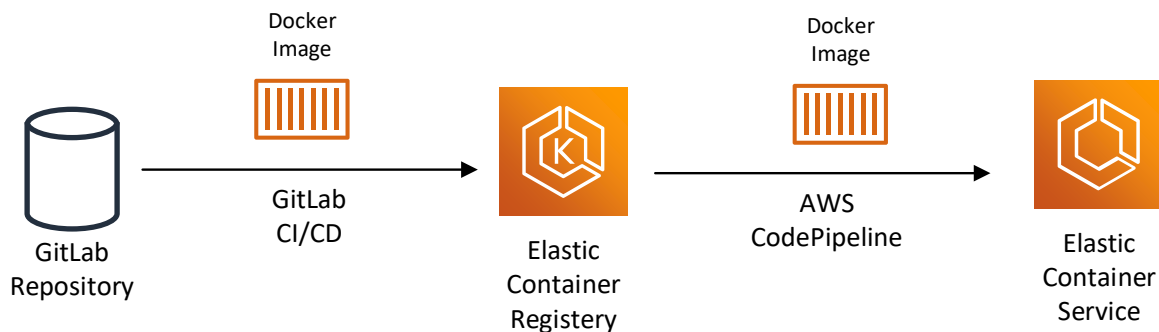

Figure 2 – ECR/ECS pipeline architecture

The GitLab CI/CD configuration file (.gitlab-ci.yml) is presented in Appendix C.

One of the options of the Elastic Container Service is the use of a health check. The concept is that the service periodically sends an HTTP request to our server to check its health. If a successful response is not received within a period of time, our server is restarted. The NGINX was configured to respond to a HTTP request at the */nginx-health* path. See the *default.conf* file presented in Appendix B for the details.

![SOFWERX logo]

CAC Micro-Service (Docker)

The second version of the solution changed from a reverse proxy server to a micro-service that performed a CAC validation check. The web client now issues a HTTPS request to the micro-service. If TLS client certification handshake is successful, a status of 200 along with the message "The CAC is valid." Is returned in the HTTPS response. The web client then logins into the system with generic CAC user credentials. If the handshake fails, the web client prompts the user to login using his or her specific credentials. A diagram of this architecture is shown below.
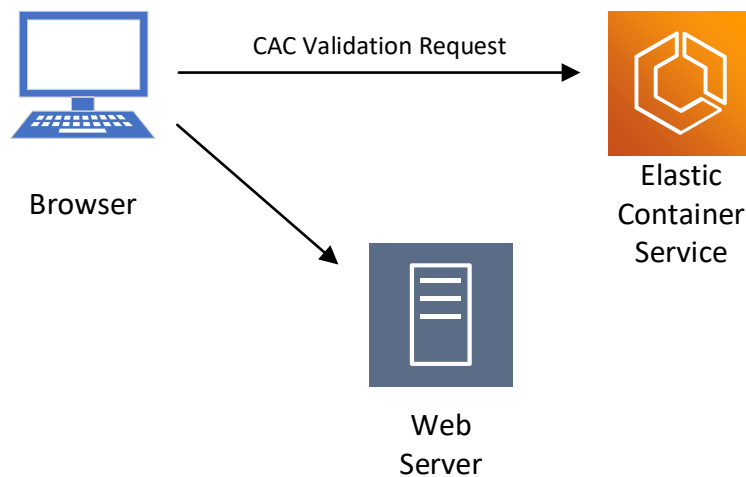

Figure 3 – Docker-based micro-service

The migration to a micro-service solution required a change to the web client code along with slight modification to the NGINX configuration file. The other artifacts are the same as in the first version. The *default.conf* file for the NGINX server is presented in Appendix D.

CAC Micro-Service (Serverless)

SOFWERX is currently migrating its public facing website from WordPress to the AWS serverless stack. With this effort, we convert the CAC validation service from Docker/ECS to the serverless technologies. The interface between the web client and the micro-service remains the same. The service is now implemented with a simpler, more scalable, and less expensive architecture. The AWS services consist of the Api Gateway, Lambda, and S3.

- API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, CORS support, authorization and access control, throttling, monitoring, and API version management [6].

- AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers, creating workload-aware cluster scaling logic, maintaining event integrations, or managing runtimes [7].

- Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance [8].

This version of the service is based on an AWS serverless project previously completed at SOFWERX. The project created a working web application that included a web client using JavaScript and React. The web API service comprised of API Gateway and Lambda with Python. The Fast Path to AWS Serverless Applications whitepaper describes the project in detail.

The micro-service exhibits the same external behavior as the Docker-based version. The request to and the response from this service remain unchanged. The web client issues a HTTPS request to the API Gateway. The gateway is configured to require client certificates and has a path to the DoD public certificates located in an S3 bucket. If the CAC is valid, the gateway then invokes the Lambda service which is a simple Python function. A response is returned with a status of 200 along with the body with the text "The CAC is valid." If the request is not successful, the gateway returns the appropriate status code. A diagram of the architecture is shown below:
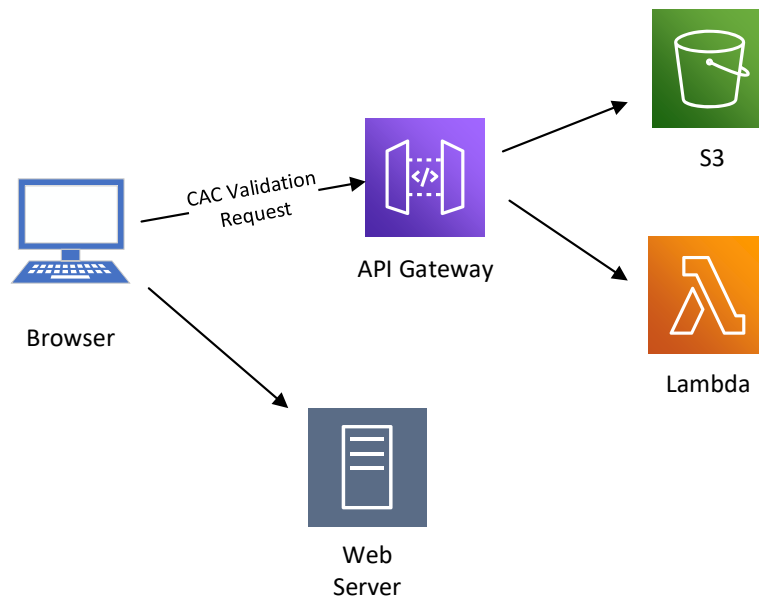

Figure 4 – Serverless-based micro-service

The build and deployment of this solution relies on the GitLab CI/CD pipeline service although it is less complex than the one for the Docker-based solutions. A diagram of the pipeline is shown below. The GitLab CI/CD configuration file (.gitlab-ci.yml) is presented in Appendix E.
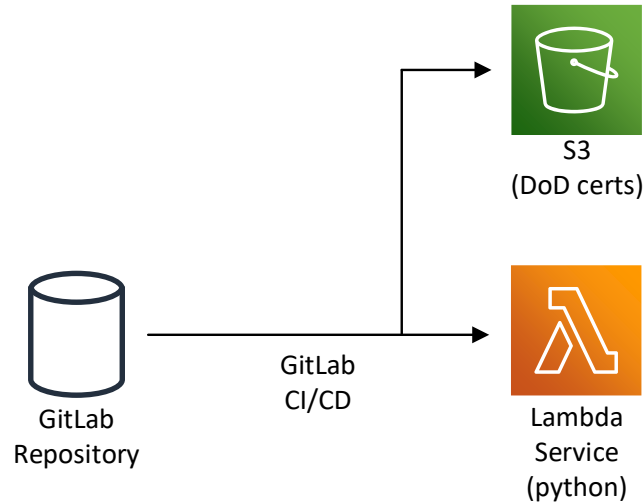
SOFWERX

Figure 5 – Serverless pipeline architecture

There are a few manual steps required before the API Gateway is fully configured and the DoD certificate file is ready to use. Once these steps are completed, the GitLab CI/CD pipeline is fully functional. I found the following two references to be extremely helpful:

[Introducing mutual TLS authentication for Amazon API Gateway](#)

[Configuring mutual TLS authentication for a REST AP](#)

Here are some useful notes that were gleaned along the way:

- Once the DoD certificate bundle is uncompressed, the p7b file must be converted to the pem format in order to be read by the API Gateway. The following command will perform the conversion. Once the *DoD_Cas.pem* is created, copy it to the project home directory in the GitLab repository so the CI/CD pipeline can access it.

  ```
  openssl pkcs7 -in Certificates_PKCS7_v5.8_DoD.pem.p7b -print_certs -out
  DoD_CAs.pem
  ```

- The API Gateway requires an Amazon Certificate Management (ACM) issued certificate regardless of where your domain name was created. During the request process, you will be prompted to select either a DNS modification option or an email option to verify the ownership of the domain name. The email option has proven to be a much faster and more reliable than the DNS record option.

- In the API Gateway, the default Authorization for the Method Request is IAM. For the Method Request of our resource, the Authorization needs to be set to NONE. Since the API Gateway is configured for TLS Client Certificates, it will reject any requests that do not have a valid CAC before it gets to the Method Request. At this point, we want the gateway to invoke our Lambda code.

- Be sure to disable the URL endpoint in the API Gateway. I usually test with the endpoint, disable it, and test again to ensure that the request fails.

- Be sure to deploy the API after any updates are made.

- Once everything is working, a HTTPS request is issued with a valid CAC, a response will be returned with a status code of 200 and a body containing the text "The CAC is valid.". It the CAC is invalid or not present, a failure of connection reset condition will be reported by the client.

Summary

SOFWERX incorporated the Common Access Card, as issued from the Department of Defense, into its web application in October of 2019. It has been a successful and popular enhancement. The users can quickly access the protected online information without having to login and the protected data is available only to valid CAC holders. As with most software-based projects, this project evolved over time and is now a micro-service implemented by AWS serverless technologies. I believe this is the final iteration of the service but, then again, who really knows?

Who We Are

SOFWERX is a non-profit entity that accelerates evolution of the Warfighter through technology discovery, engagement, development, and transition. SOFWERX was created under a Partnership Intermediary Agreement, established in September of 2015, between DEFENSEWERX and the United States Special Operations Command (USSOCOM).

Jim Ladd is a Senior Software Architect and IT Manager at SOFWERX where he leads the software engineering, web development, and system administration teams. Jim has been developing software solutions for over 35 years. Before joining SOFWERX in 2019, Jim was CEO and Principal Consultant at Wazee Group, a niche consulting company, for 20 years. His LinkedIn profile link is
https://www.linkedin.com/in/jim-ladd/

References

[1] Department of Defense, "DoD Common Access Card," [Online]. Available: https://www.cac.mil/common-access-card/. [Accessed 13 07 2021].

[2] Defense Information Systems Agency, "DoD CYBER EXCHANGE," [Online]. Available: https://public.cyber.mil/pki-pke/pkipke-document-library/. [Accessed 2021 15 07].

[3] NGINX, "What is a Reverse Proxy Server?," [Online]. Available: https://www.nginx.com/resources/glossary/reverse-proxy-server/.

[4] M. Pyne, "NGINX CAC," [Online]. Available: https://github.com/mpyne-navy/nginx-cac.

[5] Amazon Web Services, "Amazon Elastics Container Service," [Online]. Available: https://aws.amazon.com/ecs/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc&ecs-blogs.sort-by=item.additionalFields.createdDate&ecs-blogs.sort-order=desc.

[6] Amazon Web Services, "Amazon API Gateway," [Online]. Available: https://aws.amazon.com/api-gateway/.

[7] Amazon Web Services, "AWS Lambda," [Online]. Available: https://aws.amazon.com/lambda/.

[8] Amazon Web Services, "Amazon S3 - Object storage built to store and retrieve any amount of data from anywhere," [Online]. Available: https://aws.amazon.com/s3/.

Appendix A – Dockerfile for Reverse Proxy Server

```
FROM alpine:3.10

RUN apk update && apk add nginx && mkdir -p /run/nginx /www/data

# Provided in this Docker package, and relatively simple configs
COPY default.conf /etc/nginx/conf.d/default.conf

# server cert
COPY sofwerx_org_keys/bundle_all.pem /etc/nginx/bundle_all.pem

# server privkey
COPY sofwerx_org_keys/sofwerx_org_key.pem  /etc/nginx/sofwerx_org_key.pem

# The Makefile will generate DoDRoots.crt (the DoD root + intermediate certs
# concatenated) by downloading the certs from the cyber.mil IASE website.
# NGINX needs these certs to setup the CA it trusts for client
# authentication.
#
# See also the default.conf's configuration where we permit NGINX to have
# multiple intermediate certs.
COPY DoDRoots.crt /etc/nginx

EXPOSE 443 80

ENTRYPOINT ["/usr/sbin/nginx", "-q", "-g", "daemon off;"]
```

Appendix B – NGINX Configuration for Reverse Proxy Server

```
error_log stderr debug;

server {
    listen 443 ssl http2 default_server;
    listen [::]:443 ssl http2 default_server;

    server_name localhost;

    ssl_certificate     /etc/nginx/bundle_all.pem;
    ssl_certificate_key /etc/nginx/sofwerx_org_key.pem;
    ssl_protocols       TLSv1.1 TLSv1.2;
    ssl_ciphers         HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers on;

    ssl_verify_client on;
    ssl_verify_depth 4; # Allow intermediate CAs
    ssl_client_certificate /etc/nginx/DoDRoots.crt;

    add_header Strict-Transport-Security max-age=15768000;

    # Inform the proxyed app who the user who that SSL-terminated
    add_header X-Subject-DN $ssl_client_s_dn always;
    add_header X-Client-Verified $ssl_client_verify always;

    root /www/data;

    location / {
        proxy_pass https://sofwerx.org/;
        proxy_set_header Accept-Encoding "";
        proxy_set_header Proxy-Client-CAC-Type "SOCOM_SOFWERX_PROJECTS";
        sub_filter "https://sofwerx.org" "https://projects.sofwerx.org";
        sub_filter_once off;
    }
}

server {
    server_name   localhost;
    listen        80;

    error_page    500 502 503 504  /50x.html;

        location /nginx-health {
            return 200 "nginx is healthy\n";
            add_header Content-Type text/plain;
        }

    location / {
        root      html;
    }
}
```

Appendix C – GitLab CI/CD Configuration for AWS ECR

```
image: docker:latest

variables:
  REPOSITORY_URL: 094013511888.dkr.ecr.us-east-1.amazonaws.com/sofwerx-cac-
reverse-proxy:latest

services:
- docker:dind

before_script:
  - apk add --no-cache curl jq python3 py3-pip
  - pip install awscli

stages:
  - build

build:
  stage: build
  script:
    - $(aws ecr get-login --no-include-email --region us-east-1)
    - docker build -t $REPOSITORY_URL .
    - docker push $REPOSITORY_URL
  only:
    - master
  tags:
    - docker
```

Appendix D – NGINX Configuration for Micro-Service

```
error_log stderr debug;

server {
    listen 443 ssl http2 default_server;
    listen [::]:443 ssl http2 default_server;

    server_name localhost;

    ssl_certificate     /etc/nginx/bundle_all.pem;
    ssl_certificate_key /etc/nginx/sofwerx_org_key.pem;
    ssl_protocols       TLSv1.1 TLSv1.2 TLSv1.3;
    ssl_ciphers         HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers on;

    ssl_verify_client on;
    ssl_verify_depth 4; # Allow intermediate CAs
    ssl_client_certificate /etc/nginx/DoDRoots.crt;

    root /www/data;

    location / {
      default_type text/html;

      # Preflighted requests
      if ($request_method = OPTIONS ) {
        add_header "Access-Control-Allow-Origin" *;
        add_header "Access-Control-Allow-Methods" "GET, POST, OPTIONS, HEAD";
        add_header "Access-Control-Allow-Headers" "Authorization, Origin, X-
Requested-With, Content-Type, Accept";
        return 200;
      }

      # Simple requests
      if ($request_method ~* "(GET|POST)") {
        add_header "Access-Control-Allow-Origin" *;
        add_header "Access-Control-Allow-Methods" "GET, POST, OPTIONS, HEAD";
        add_header "Access-Control-Allow-Headers" "Authorization, Origin, X-
Requested-With, Content-Type, Accept";
      }

      return 200 "The CAC is valid.\n";
    }
}
```

Appendix E – GitLab CI/CD Configuration for AWS API Gateway/Lambda

```yaml
image: nikolaik/python-nodejs:python3.8-nodejs12

stages:
    - deploy

production:
    stage: deploy
    before_script:
        - python --version
        - pip3 install awscli --upgrade
        - pip install boto3
        - npm install serverless -g
        - npm install aws-sdk --save-dev
        - npm install uuid@7.0.3 --save
        - npm install --save serverless-python-requirements
        - cd swx-cac-service
        - cd infrastructure
        - npm install @serverless-stack/cli --save-exact
    script:
        - npx sst build
        - npx sst deploy
        - pwd
        - cd -
        - cd services/swx_cac
        - serverless deploy -v
        - cd $CI_PROJECT_DIR
        - aws s3 cp $CI_PROJECT_DIR/DoD_CAs.pem s3://swx-ca-truststore/DoD_CAs.pem
    environment: production
```