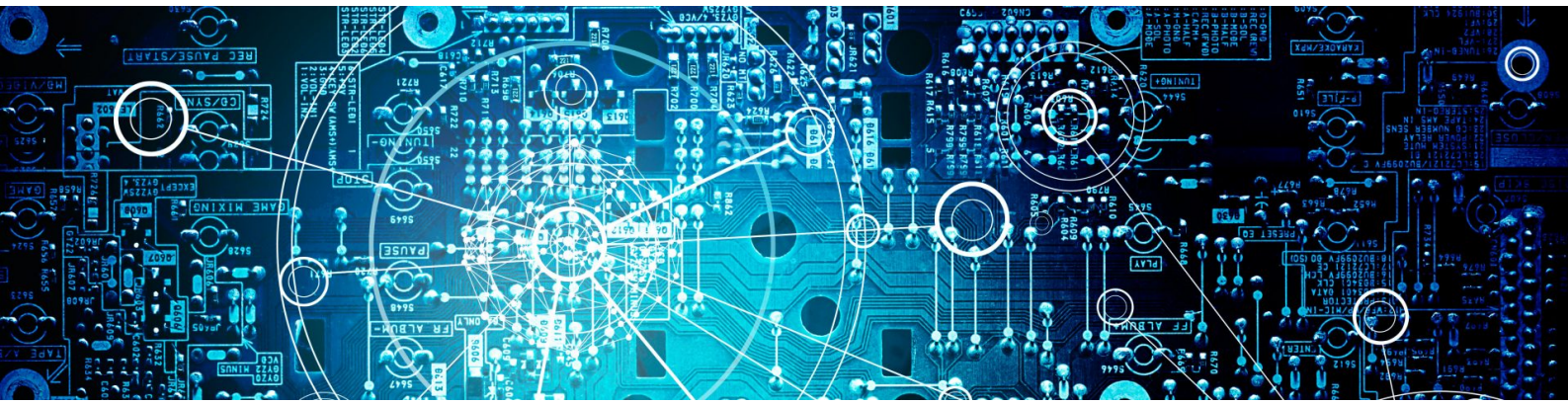
KUDELSKI SECURITY | RESEARCH

The Latest News from Research at Kudelski Security



FIRST STEPS TOWARDS A ZERO TRUST ARCHITECTURE

August 4, 2020 Bojan Zelic Network, Network security Leave a comment

Hybrid and multi-cloud infrastructures are a real challenge in term of security and user accesses management. Traditional solutions like VPNs are usually not adapted for such scenarios. They could still work but at the expense of building a complex (and costly) web of

interconnections and where micro-segmentation of accesses would be complex to manage.

This blog post is the first one of a series to document our journey towards Zero Trust at Kudelski Security. Today we will focus on securing the exposition of internal application through Cloudflare Access leveraging some tools and code that we just open-sourced.

Zero Trust in two lines

Behind the buzzword, Zero Trust is not about making a system trusted; it's about eliminating the trust that we would originally put on the network. So whenever your users are connecting to your application from the corporate network or from their home WiFi, they will follow the same authentication and authorization workflow.

Identity is one of the building-blocks of a Zero Trust Architecture (Identity Based Security) that's why Zero Trust is tightly linked to Identity and Accesses Management (IAM) and device-posture-management but that's probably worth another blog post.

If you want to read more on that topic:

- Zero Trust a Global Perspective
- What is a Zero Trust Architecture?
- BeyondCorp model
- BeyondCorp

Cloudflare Access

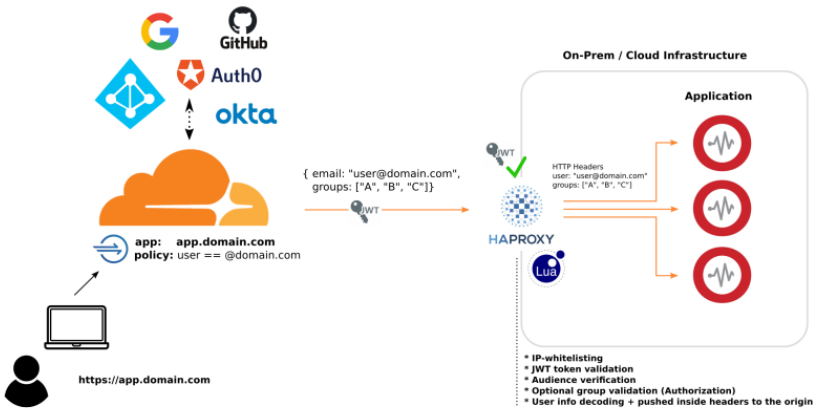
Cloudflare Access is a SaaS Identity-aware-Proxy (IaP) developed by Cloudflare that can help companies willing

to switch to a Zero Trust model by providing a platform-agnostic solution that covers most use-cases.

Exposing applications to Cloudflare Access

Because Cloudflare Access is a SaaS product that is not physically linked to datacenters, we needed to find a way to securely expose some applications to Cloudflare CDN over the public Internet. There's multiple ways and technologies to achieve that (including Argo tunnels) but the solution that we choose and implemented for this particular use-case leverages HAProxy (a TCP/HTTP load-balancer) which is a product that we were already familiar with internally.

We've added some glue on top of HAProxy, augmenting its functionalities with its LUA engine by cryptographically validating requests coming from Cloudflare Access applications. (on top of doing Layer 4 network validations).



Step-by-step implementation

This post is a step-by-step guide to implement JSON Web Token (JWT) based authentication with HAProxy & Cloudflare Access:

haproxy-cloudflare-jwt-validator is a lua script for HAProxy that validates JWT from a Cloudflare JWKS endpoint. It's able to handle authorization and authentication because it validates each request to make sure that it's associated with the correct Cloudflare Access Application. It is additionally able to pass any custom information from the JSON Web Token (like SAML group information) to HAProxy. It also enables the security of our legacy applications by adding a layer of strong authentication (external IdP + MFA) and authorization on top of them.

Last but not least, this project is about HAProxy in a Cloudflare Access context but the code and logic can be applied to anything that uses JWT-based authentication.

What are JSON Web Tokens

JSON web tokens are an open standard for transmitting information between 2 parties as a JSON object. These tokens can contain user, group and other information. They can be trusted because they can be verified since they are digitally signed. Cloudflare signs their JWT tokens with SHA-256 using a private & public key pair.

What is a JWKs?

A JSON Web Key set is a JSON structure that exposes public keys and certificates that are used for validating JSON Web Tokens. Cloudflare exposes their JWKs at https://YourAuthenticationDomain/cdn-cgi/access/cert. Cloudflare additionally rotates the certificates they use at this endpoint on a regular basis.

Why should we validate JSON Web Tokens?

Cloudflare recommends whitelisting cloudflare IPs and validating the JWT for the requests that come through. Only whitelisting IPs & failing to validate JWT means that an attacker can simply add a DNS entry in their Cloudflare configuration to bypass your IP whitelist. You need to make sure to validate the JWT token in order to prevent access from a non-authorized Cloudflare account and to ensure that nothing goes past your Access Application rules.

Demo – Signing & Validating your own JWT Locally

We have provided an example here for signing a JSON web token and validating the token

Dependencies

The jwt_test.sh command requires a few dependencies that need to be installed on your system

- Docker
- Docker-compose
- jq

```
sudo apt-get install jq (Debian/Ubuntu)
sudo yum install jq (CentOS)
brew install jq (OSX)
```

- jwtgen

```
npm install -g jwtgen
```

Run the Test

To run the example... simply run the script ex:

```
$./jwt_test.sh
```

```
Generating Private Key & Certificate:
Generating a RSA private key
.....................................
 ..............++++
.....................................
 ............++++
writing new private key to 'certs/privat
-----

Adding Certificate to JWKS Endpoint:
done
Starting example_debug_http_listener_1
Starting example_cloudflare_mock_1
Starting example_haproxy_cloudflare_jwt_

CURL Response with Bad Cf-Access-Jwt-Ass
<html><body><h1>403 Forbidden</h1>
Request forbidden by administrative rule
</body></html>

CURL Response with Valid Cf-Access-Jwt-A
{
  "path": "/",
  "headers": {
    "host": "localhost:8080",
    "user-agent": "curl/7.58.0",
    "accept": "*/*",
    "cf-access-jwt-assertion": "eyJ0eXAi
 iJ9.eyJpYXQiOjE1OTMyMDQ4NTgsImF1ZCI6WyI
 DkwYWJjZGUxMjM0NTY3ODkwYWJjZGUiXSwiZW1i
 wuY29tIiwic3ViIjoiMTIzNDU2Nzg5MCIsIm5hb
 0cnVlLCJpc3MiOiJodHRwOi8vY2xvdWRmbGFyZV
 LCJleHAiOjM5OTMyMDQ4NTgsInR5cGUiOiJhcHA
 TExMTExMTExIiwiY3VzdG9tIjp7fX0.xq1KyxFi
 nvQ4yqs8MFGTqERp1ggTxF99ieBZ_PDg79jL6NO
 H-NWOsOYMg4KCs7vkj0_g5eforln8aFdCEG1ToA
 RjHgQ692TpwPDdzFsllty4xLfzSbePpEira5pNv
 N4RI7NltUxlroQUJDnMIUA7BEVMFsRgqpNpWE3v
 QuDH7htV-uuCaWqwPm4ke4PDY2w3Hjvds5Zne5S
 3nzmKd_yMQaORRrX9-4lcOWmUoQ1E3pbLcLSQtj
 b58WrJOrQwo3-jZON1PReCnuZArsfqoE8qPbPTI
 QtHVZMDC3GlmDi9RJaribjc0sWbJrPCNqMmdl3c
 ybG1pTb7Tefd98GamEwhIpENW7z9YF2jPcNeZ8I
 nAy0wzUYMRPKiOVVu1PaZQNU",
    "x-forwarded-for": "172.20.0.1",
    "connection": "close"
  },
  "method": "GET",
  "body": "",
  "fresh": false,
  "hostname": "localhost",
  "ip": "172.20.0.1",
  "ips": [
    "172.20.0.1"
  ],
  "protocol": "http",
  "query": {},
  "subdomains": [],
```

```
    "xhr": false,
    "os": {
      "hostname": "08afc040418e"
    },
    "connection": {}
Stopping example_haproxy_cloudflare_jwt_
Stopping example_debug_http_listener_1
Stopping example_cloudflare_mock_1
```

How it works

To start off, we generate a private key and a certificate with openssl. We stick with default values just because this is a demo scenario and it doesn't matter for this test.

```
openssl req -new -newkey rsa:4096 -days 
    -subj "/C=US/ST=Denial/L=Springfield,
    -keyout certs/private.key  -out certs
```

We then take the certificate that was generated, and expose it to the JWKS endpoint (we are running a simple python server that exposes files just for demo purposes).

```
$ CERT=$(cat certs/certificate.pem)
$ jq -n --arg cert "$CERT" '{public_certs
```

We then use a custom claim to generate a JWT token.

```
CLAIM='{
  "aud": [
    "1234567890abcde1234567890abcde1234
  ],
  "email": "random-email@email.com",
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iss": "http://cloudflare_mock",
  "iat": 1593204858,
  "nbf": 1593204858,
  "exp": 3993204858,
  "type": "app",
  "identity_nonce": "11111111111",
  "custom": {}
}'
```

```
JWT_TOKEN=$(jwtgen -a RS256 -p certs/pr
curl -H "Cf-Access-Jwt-Assertion: ${JWT
```

The jwtverify.lua script then gets triggered via the defined HAProxy backend. It decodes the token, validates the algorithm, signature, expiration, issuer, and audience.

Installing the JWT validation script

Dependencies

The haproxy-cloudflare-jwt-validation lua script requires a few dependencies that need to be installed on your

Debian/Ubuntu Instructions:

```
sudo apt install lua5.3 liblua5.3-dev wg
sudo mkdir -p /usr/local/share/lua/5.3
```

haproxy-lua-http:

```
wget https://github.com/haproxytech/hapr
tar -xf master.tar.gz -C /usr/local/shar
cp /usr/local/share/lua/5.3/haproxy-lua-
```

rxi/json:

```
wget https://github.com/rxi/json.lua/arc
tar -xf v0.1.2.tar.gz -C /usr/local/shar
ln -s /usr/local/share/lua/5.3/json.lua-
```

wahern/luaossl:

```
wget https://github.com/wahern/luaossl/a
tar -xf rel-20190731.tar.gz -C /usr/loca
cd /usr/local/share/lua/5.3/luaossl-rel-
make install
```

diegonehab/luasocket

```
wget https://github.com/diegonehab/luasoc
tar -xf master.tar.gz -C /usr/local/share
cd /usr/local/share/lua/5.3/luasocket-mas
make clean all install-both LUAINC=/usr/
```

Once the dependencies are installed install the latest release of the plugin:

https://github.com/kudelskisecurity/haproxy-cloudflare-jwt-validator/releases/latest

```
tar -xf haproxy-cloudflare-jwt-validator-
ln -s /usr/local/share/lua/5.3/haproxy-cl
ln -s /usr/local/share/lua/5.3/haproxy-cl
```

Enabling the plugin

Once the lua script is installed... the plugin can be enabled by enabling the following configuration options in /etc/haproxy/haproxy.cfg (replace test.cloudflareaccess.com with your JWT issuer)

```
global
  lua-load  /usr/local/share/lua/5.3/jwt
  setenv  OAUTH_HOST     test.cloudflare
  setenv  OAUTH_JWKS_URL https://|cloud
  setenv  OAUTH_ISSUER   https://"${OAU

backend cloudflare_jwt
  mode http
  default-server inter 10s rise 2 fall 
  server "${OAUTH_HOST}" "${OAUTH_HOST}"

resolvers dnsresolver
  nameserver dns1 1.1.1.1:53
  nameserver dns2 1.0.0.1:53
  resolve_retries 3
  timeout retry 1s
  hold nx 10s
  hold valid 10s
```

Authentication validation with JWT

In addition to validating the JWT token with haproxy-cloudflare-jwt-validator, it's recommended to block and only whitelist cloudflare IPs for your publicly exposed endpoint in your firewall.

The haproxy-cloudflare-jwt-validator script performs authentication on your pre-defined backend or frontend.

For example:

```
backend my_jwt_validated_app_backend
  mode http
  http-request deny unless { req.hdr(Cf-
  http-request set-var(txn.audience) st
  http-request lua.jwtverify
  http-request deny unless { var(txn.au
  server haproxy 127.0.0.1:8080
frontend my_jwt_validated_app_frontend
  bind *:80
  mode http
  use_backend my_jwt_validated_app_back
```

Using the configuration above... when a HTTP request comes through to port 80 the following checks are performed:

- Validates that the Cf-Access-Jwt-Assertion Header is set
- Validates that the Algorithm of the token is RS256
- Validates that the at least one public key from the JWKS certificates match the public key for the signed JSON Web Token
- Validates that the token is not expired
- Validates that the Issuer of the token matches the predefined issuer from the OAUTH_ISSUER ENV variable
- Validates that the audience tag from the JWT token matches the audience tag that we expect for this backend

Assuming that all of the above checks pass... the request is then allowed. If any of the above checks fail. HAProxy

will respond with a 403 (Unauthorized) error.

Authorization validation with JWT

App authorization runs under the assumption that you can pass information from Cloudflare such as group information, permissions, etc ... as part of the JWT token.

For example, when you add an SAML Identity Provider to Cloudflare Access... you can define a list of SAML attributes that will get included as part of the encoded JSON Web Token. These attributes are passed through as variables that can be used in headers, HAProxy authentication, or even other lua scripts.

When you configure SAML attributes successfully, you will see them included as part of the decoded JSON Web Token as part of the 'custom' key. Ex:

```
{
  ...
  "email": "random-email@email.com",
  "name": "John Doe",
  "custom": {
    "http://schemas/groups": [
      "application_admin",
      "application_group1",
      "application_group2",
      "application_group3"
    ]
  }
}
```

The haproxy-cloudflare-jwt-validate script will take the keys defined in "custom" and declare haproxy variables with them. (This will strip out special characters and replace them with an underscore)

For example: http://schemas/groups becomes txn.http___schemas_groups

App-Based Header Authorization

Now that we've defined our variable txn.http___schemas_groups; You can then pass this variable through to the headers. Every request that comes through will pass this header information to the backend by simply using set-headers.

```
backend my_jwt_validated_app_backend
  mode http
  http-request deny unless { req.hdr(Cf-A
  http-request set-var(txn.audience) str
  http-request lua.jwtverify
  http-request deny unless { var(txn.auth
  http-request set-header custom-groups %
  server haproxy 127.0.0.1:8080
```

Your app can read the headers, create your user based off of these headers, and assign it group information.

HAProxy Based Authorization

Another alternative we can do, is validate the presence of a user's group via HAProxy directly. For example in the following scenario, HAProxy will not grant a user access unless they belong to the application_admin group.

```
backend my_jwt_validated_app_backend
  mode http
  http-request deny unless { req.hdr(Cf-A
  http-request set-var(txn.audience) str
  http-request lua.jwtverify
  http-request deny unless { var(txn.auth
  http-request deny unless { var(txn.http
  server haproxy 127.0.0.1:8080
```

References and links

- Zero Trust a Global Perspective
- What is a Zero Trust Architecture?
- BeyondCorp model

- BeyondCorp
- JWT
- haproxy-lua-jwt