terraform_如何使用Terraform仅使用私有IP部署Cloud SQL DB

翻译

weixin_26722031 于 2020-07-30 15:31:29 发布 195 收藏
原文链接：https://medium.com/swlh/how-to-deploy-a-cloud-sql-db-with-a-private-ip-only-using-terraform-e184b08eca64

terraform

This is a writeup of how I configured a Cloud SQL database to have only a private IP address. The db is unreachable from the public internet except for a tiny hole in the private network that allows me to connect from my laptop. All traffic to the db is encrypted and routed over Google's internal network. This is a big win for data security, but it was a pain to set up. I'm writing this in the hope you can avoid some of my mistakes.

这是我将Cloud SQL数据库配置为仅具有私有IP地址的文章。 除了专用网络中的一个小Kong使我可以从笔记本电脑进行连接之外，该数据库无法从公共Internet到达。 到数据库的所有流量都经过加密，并通过Google的内部网络进行路由。 对于数据安全而言，这是一个巨大的胜利，但建立起来却很痛苦。 我写这篇文章是希望您能避免我的一些错误。

If you're in a hurry and already know Terraform and GCP well, you can skip the explanation and jump straight to the Terraform files on GitHub.

如果您很着急并且已经很熟悉Terraform和GCP，则可以跳过说明， 直接跳转到GitHub上的Terraform文件 。

总览 (Overview)

- Prerequisites
  先决条件
- Configuring a VPC with a private IP address range
  使用专用IP地址范围配置VPC
- Configuring a Cloud SQL db to have a private IP address only
  将Cloud SQL数据库配置为仅具有私有IP地址
- Using Cloud SQL Proxy to open a tiny hole in your private network so you can actually access the db
  使用Cloud SQL代理在您的专用网络中打开一个小Kong，以便您实际上可以访问数据库
- Connecting to the database through SSH
  通过SSH连接到数据库
- Conclusion
  结论

先决条件 (Prerequisites)

Skip this section if you already have `terraform` connected to your GCP project and all the required Google APIs enabled.

如果您已经将`terraform`连接到GCP项目并启用了所有必需的Google API，请跳过此部分。

You'll need the following:

您将需要以下内容：

A GCP project

一个GCP项目

`gcloud` installed, up-to-date, logged in, and connected to your project

`gcloud` 已安装， 最新 ， 已登录并已连接到您的项目

I ran these commands on macOS:

我在macOS上运行了以下命令：

```
> brew cask install google-cloud-sdk
> gcloud components update
> gcloud auth login
> gcloud config set project <my-project>
> gcloud config list
```

3. All required Google Cloud APIs enabled

3.已启用所有必需的Google Cloud API

One of the annoying things about Google Cloud is that you can't use their services right away. If you have a fresh project, you have to enable a bunch of APIs first. For example, you need to enable the Google Compute API before you can spin up a VM. A Google eng told me this extra step was "mostly for you to recognize that there's a cost." I guess that makes sense, but it does add some hassle when you're trying to automate infra provisioning.

关于Google Cloud的烦人的事情之一就是您不能立即使用他们的服务。 如果您有一个新项目，则必须首先启用大量API。 例如，您需要先启用Google Compute API，然后才能启动VM。 一位Google工程师告诉我，这一额外步骤"主要是让您认识到要付出代价的"。 我想这是有道理的，但是当您尝试自动化基础设施配置时，确实增加了一些麻烦。

We need to enable six APIs for this project. We *could* use Terraform to enable them, but I strongly advise against this. Terraform has no knowledge of the dependencies between resources and APIs. `terraform apply` will often fail because it tried to create a resource before the right API was enabled, or because it tried to enable APIs in the wrong order. Unless you want to map all these relationships yourself by adding `depends_on` to most of your resources, **just use `gcloud`**.

我们需要为此项目启用六个API。 我们*可以*使用Terraform启用它们 ，但是我强烈建议您不要这样做 。 Terraform不了解资源和API之间的依赖关系。 `terraform apply`通常会失败，因为它尝试在启用正确的API之前创建资源，或者因为尝试以错误的顺序启用API。 除非您想通过向大多数资源中添加`depends_on`来自己映射所有这些关系，否则请**使用 `gcloud`** 。

```
> gcloud services enable \
    cloudresourcemanager.googleapis.com \
    compute.googleapis.com \
    iam.googleapis.com \
    oslogin.googleapis.com \
    servicenetworking.googleapis.com \
    sqladmin.googleapis.com
```

Yes, it sucks that your infra code won't be 100% Terraform, but a one-line bash script isn't the worst. This is a one-time task. Once the APIs are enabled, leave them enabled.

是的，很糟糕，您的基础代码不会是100％Terraform的，但是单行bash脚本并不是最差的。 这是一项一次性的任务。 启用API后，将其保持启用状态。

4. A Terraform Cloud workspace

4. Terraform Cloud工作空间

For this project, I'm using Terraform Cloud. This is a service from Hashicorp with a few important features over vanilla Terraform. The most important feature is *remote execution*. Terraform works by reading code from your `.tf` files and converting it into calls to your cloud provider's API. When you run `terraform apply` on your laptop with a flaky Wi-Fi connection, you're trusting that all network requests will succeed. If they don't, your cluster could be left in a half-deployed state. You can try tearing down the cluster, or you can try finishing the deploy. Both options have a high probability of failing and costing you time. With remote execution, the `apply` is being run on a highly available cloud server managed by Hashicorp.

对于这个项目，我正在使用Terraform Cloud 。 这是Hashicorp提供的一项服务，具有优于香草Terraform的一些重要功能。 最重要的功能是*远程执行* 。 Terraform通过从`.tf`文件中读取代码并将其转换为对云提供商的API的调用来工作。 当您通过`terraform apply` Wi-Fi连接在笔记本电脑上运行`terraform apply`时，您会相信所有网络请求都会成功。 如果不这样做，您的集群可能会处于半部署状态。 您可以尝试拆除群集，也可以尝试完成部署。 两种选择都极有可能导致失败并浪费您时间。 通过远程执行， `apply`程序将在由Hashicorp管理的高可用性云服务器上运行。

Another key feature of Terraform Cloud is secure, remote storage of state files. Terraform state files are created or mutated whenever we run an `apply` or `destroy`. They contain secrets and need to be stored safely. I'd rather not have the state file on my local machine, where I'm responsible for keeping it safe.

Terraform Cloud的另一个关键功能是状态文件的安全，远程存储 。 每当我们运行`apply`或`destroy`时，都会创建或更改Terraform状态文件。 它们包含秘密，需要安全存储。 我宁愿不在本地机器上保存状态文件，而要负责安全状态文件。

Terraform Cloud is free for up to five users. I strongly recommend it.

Terraform Cloud 可供多达五个用户免费使用 。 我强烈推荐。

7. A GCP service account for Terraform Cloud

7. Terraform Cloud的GCP服务帐户

Once you've created a `terraform-cloud` service account in GCP, take the entire JSON key, minify it, and save it as the `GOOGLE_CREDENTIALS` env var in your workspace. It's a secret, so make sure to check the "Sensitive" box when you set it. You must minify the JSON. Terraform Cloud will not allow you to set an env var with newlines in it.

在GCP中创建了`terraform-cloud`服务帐户后，请使用整个JSON密钥，将其最小化，然后将其另存为工作区中的`GOOGLE_CREDENTIALS` 。 这是个秘密，因此在设置时请务必选中"敏感"框。 您必须缩小JSON。 Terraform Cloud不允许您设置带有换行符的环境变量。

Image for post

Now that Terraform Cloud is connected to your GCP project, we can finally start this journey. To keep things short, I'm going to omit module organization, variables, and outputs. I'll just focus on the resources. If you want the rest, you can see the complete code on GitHub.

现在Terraform Cloud已连接到您的GCP项目，我们终于可以开始这一旅程。 为了简短起见，我将省略模块的组织，变量和输出。 我只关注资源。 如果需要其余内容，可以在GitHub上查看完整的代码 。

使用专用IP地址范围配置VPC (Configuring a VPC with a private IP address range)

First, we need to create a VPC.

首先，我们需要创建一个VPC。

```
resource "google_compute_network" "vpc" {
  name                    = var.name
  routing_mode            = "GLOBAL"
  auto_create_subnetworks = true
}
```

Then, we need to allocate a block of private IP addresses.

然后，我们需要分配一个私有IP地址块。

```
resource "google_compute_global_address" "private_ip_block" {
  name          = "private-ip-block"
  purpose       = "VPC_PEERING"
  address_type  = "INTERNAL"
  ip_version    = "IPV4"
  prefix_length = 20
  network       = google_compute_network.vpc.self_link
}
```

We don't specify the exact address range. Google will select the range for us. We only need to specify how many addresses we want. A prefix length of 20 will create around four thousand IP addresses. That's plenty.

我们没有指定确切的地址范围。 Google将为我们选择范围。 我们只需要指定我们想要多少个地址。 前缀长度为20将创建约4000个IP地址。 够了

Then we need to enable something called private services access. This is what allows our instances to communicate exclusively using Google's internal network.

然后，我们需要启用称为私有服务访问的功能 。 这就是我们的实例可以使用Google的内部网络专门进行通信的原因。

```
resource "google_service_networking_connection" "private_vpc_connection" {
  network                 = google_compute_network.vpc.self_link
  service                 = "servicenetworking.googleapis.com"
  reserved_peering_ranges = [google_compute_global_address.private_ip_block.name]
}
```

Finally, we need to add a firewall rule to allow ingress SSH traffic.

最后，我们需要添加防火墙规则以允许入口SSH流量。

```
resource "google_compute_firewall" "allow_ssh" {
  name        = "allow-ssh"
  network     = google_compute_network.vpc.name
  direction   = "INGRESS"
  allow {
    protocol = "tcp"
    ports    = ["22"]
  }
  target_tags = ["ssh-enabled"]
}
```

The `target_tags` field is important. This firewall rule will only apply to instances that have been tagged with `"ssh-enabled"`. We'll use this tag later when we set up a special instance that proxies traffic to the db (Cloud SQL Proxy).

`target_tags`字段很重要。 此防火墙规则仅适用于已标记为`"ssh-enabled"`实例。 稍后，当我们设置一个特殊实例以代理到数据库的流量(Cloud SQL代理)时，将使用此标记。

将Cloud SQL数据库配置为仅具有私有IP地址 (Configuring a Cloud SQL db to have a private IP address only)

Now that we have an isolated network, let's put our db in it. In Terraform, creating one db requires two resources for some reason.

现在我们有了隔离的网络，让我们将数据库放入其中。 在Terraform中，由于某种原因，创建一个数据库需要两个资源。

```
resource "google_sql_database" "main" {
  name     = "main"
  instance = google_sql_database_instance.main_primary.name
}resource "google_sql_database_instance" "main_primary" {
  name             = "main-primary"
  database_version = "POSTGRES_11"
  depends_on       = [google_service_networking_connection.private_vpc_connection]  settings {
    tier              = "db-f1-micro"
    availability_type = "REGIONAL"
    disk_size         = 10  # 10 GB is the smallest disk size    ip_configuration {
      ipv4_enabled    = false
      private_network = google_compute_network.vpc.self_link
    }
  }
}resource "google_sql_user" "db_user" {
  name     = var.user
  instance = google_sql_database_instance.main_primary.name
  password = var.password
}
```

There are a few important bits here:

这里有一些重要的地方：

1. I'm creating a Postgres instance here, but it's easy to convert my Terraform code to MySQL or MS SQL Server.
   我在这里创建一个Postgres实例，但是很容易将Terraform代码转换为MySQL或MS SQL Server。

We have to explicitly state that the db depends on private services access with a `depends_on` statement. Terraform will not figure this out on its own.

我们必须使用`depends_on`语句明确声明数据库依赖于私有服务访问 。 Terraform不会自行解决这个问题。

We must set `ipv4_enabled = false` to prevent the db from getting a public IP. This doesn't stop the db from getting a private IPv4 address.

我们必须设置`ipv4_enabled = false`来防止数据库获取公共IP。 这不会阻止数据库获取私有IPv4地址。

4. The db password is a secret. It should be set on the Variables tab of Terraform Cloud.
db密码是一个秘密。 应该在Terraform Cloud的“变量”选项卡上进行设置。

Image for post

If you run `terraform apply` right now, you'll see it create a VPC and a db with only a private IP. Woohoo! We're done, right?

如果立即运行`terraform apply`，您将看到它创建了仅具有私有IP的VPC和数据库。 hoo！ 完成了吧？

Not quite. We've created a very secure db, but it's so secure that we ourselves can't access it! □ To make this db actually reachable (with `psql` or `mysql` or whatever), we need to use something called Cloud SQL Proxy.

不完全的。 我们已经创建了一个非常安全的数据库，但是它是如此安全以至于我们自己无法访问它！ □为了使该数据库实际上可访问(通过`psql`或`mysql`或其他方式)，我们需要使用称为Cloud SQL Proxy的东西。

使用Cloud SQL代理在您的专用网络中打开一个小Kong，以便您实际上可以访问数据库 (Using Cloud SQL Proxy to open a tiny hole in your private network so you can actually access the db)

What's the point of locking down the db on a private network if we're just going to open a hole in that network and let the world in? Well, we've still made a big security improvement by keeping the db instance off the public internet. Now we're going to run a special VM instance on the private network that also has a public IP address. This will serve as a middleman between the public internet and our private network. It will be hardened as much as possible so only authorized SSH users can get through. Think of it as an exceptionally fortified gate. If you've heard of a bastion box or a jump box, that's what this is. This instance will have a much smaller attack surface compared to the db instance. It will run `sshd` and a binary called `cloud_sql_proxy` and that's it.

如果我们只是要在专用网络上打开一个漏洞并让世界进入，那么锁定专用网络上的数据库有什么意义呢？ 好吧，我们仍然通过将数据库实例保留在公共Internet之外，在安全性方面进行了很大的改进。 现在，我们将在还具有公共IP地址的专用网络上运行一个特殊的VM实例。 这将充当公共互联网和我们的专用网络之间的中间人。 它将尽可能得到加强，以便只有授权的SSH用户才能通过。 可以将其视为一个异常坚固的大门。 如果您听说过堡垒箱或跳箱，那就是这样。 与db实例相比，此实例的受攻击面要小得多。 它将运行`sshd`和一个名为`cloud_sql_proxy`的二进制`cloud_sql_proxy`，仅此而已。

First, we need a service account for `cloud_sql_proxy`, so it can connect to the db.

首先，我们需要一个`cloud_sql_proxy` 服务帐户 ，以便它可以连接到数据库。

```
resource "google_service_account" "proxy_account" {
  account_id = "cloud-sql-proxy"
}resource "google_project_iam_member" "role" {
  role   = "roles/cloudsql.editor"
  member = "serviceAccount:${google_service_account.proxy_account.email}"
}resource "google_service_account_key" "key" {
  service_account_id = google_service_account.proxy_account.name
}
```

We gave the service account the Cloud SQL Editor IAM role, so it has full read-write access to the db.

我们为服务帐户赋予了Cloud SQL编辑器IAM角色，因此它具有对数据库的完全读写访问权限。

Next, we'll create the proxy instance.

接下来，我们将创建代理实例。

```
data "google_compute_subnetwork" "regional_subnet" {
  name   = google_compute_network.vpc.name
  region = "us-central1"
}resource "google_compute_instance" "db_proxy" {
  name                      = "db-proxy"
  machine_type              = "f1-micro"
  zone                      = "us-central1-a"
  desired_status            = "RUNNING"
  allow_stopping_for_update = true  tags = ["ssh-enabled"]  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
      size  = 10
      type  = "pd-ssd"
    }
  }  metadata = {
    enable-oslogin = "TRUE"
  }  metadata_startup_script = templatefile("${path.module}/run_cloud_sql_proxy.tpl", {
"db_instance_name"    = "db-proxy",
"service_account_key" = base64decode(google_service_account_key.key.private_key),
})  network_interface {
    network    = var.vpc_name
    subnetwork = data.google_compute_subnetwork.regional_subnet.self_link     access_config {}
  }  scheduling {
    on_host_maintenance = "MIGRATE"
  }  service_account {
    email = module.serviceaccount.email
    scopes = ["cloud-platform"]
  }
}
```

There's a lot to unpack here. Let's take it one block at a time.

这里有很多要解压的东西。 让我们一次取一个方块。

```
data "google_compute_subnetwork" "regional_subnet" {
  name   = google_compute_network.vpc.name
  region = "us-central1"
}
```

The VPC is a global network that covers all Google Cloud datacenters. Each region is given its own subnet. We could create the proxy instance anywhere, but for simplicity, we'll create it in the same region as the database. This data source says "give me the subnet for the `us-central1` region."

VPC是覆盖所有Google Cloud数据中心的全球网络。 每个区域都有自己的子网。 我们可以在任何地方创建代理实例，但为简单起见，我们将在与数据库相同的区域中创建它。 该数据源说："给我`us-central1`区域的子网。"

```
boot_disk {
  initialize_params {
    image = "cos-cloud/cos-stable"
  }
}
```

`cos-stable` is the latest stable version of Container-Optimized OS. This is a special Linux distro made by Google that has some properties that make it ideal for a bastion box.

`cos-stable`是Container-Optimized OS的最新稳定版本。 这是Google制作的特殊Linux发行版，其某些属性使其非常适合用作堡垒盒。

1. It's based on Chromium OS, a fast-booting, stripped-down version of Linux.
   它基于Chromium OS，Chromium OS是快速启动的精简版Linux。
2. It auto-updates itself □
   它会自动更新□
3. It can only run programs in containers.
   它只能在容器中运行程序。

So, it has a very small attack surface, and requires zero effort to keep it up-to-date with the latest security fixes.

因此，它具有很小的攻击面，并且不需要任何努力就可以使用最新的安全修复程序来使它保持最新。

```
tags = ["ssh-enabled"]
```

This tag instructs the firewall to allow inbound SSH traffic.

此标记指示防火墙允许入站SSH通信。

```
metadata = {
  enable-oslogin = "TRUE"
}
```

OS Login is an awesome service that takes away the hassle of managing SSH keys for your instances. Instead of *you* being responsible for putting all the keys on each box, Google will take care of that for you. You simply upload your public SSH key to the OS Login service one time. Then you can access any box with `enable-oslogin = TRUE` set in the instance's metadata.

OS Login是一项很棒的服务，它消除了为实例管理SSH密钥的麻烦。 Google不必为将所有键放在每个框上负责，而是由*您*来解决。 您只需一次将您的公共SSH密钥上载到OS Login服务。 然后，您可以访问在实例的元数据中设置了`enable-oslogin = TRUE`任何框。

```
> gcloud compute os-login ssh-keys add --key-file=~/.ssh/id_rsa.pub --ttl=365d
```

The command above uploads your public SSH key and sets it to expire in one year. For bonus points, you can enable 2FA on OS Login.

上面的命令将上传您的公共SSH密钥，并将其设置为在一年内到期。 对于积分，您可以在OS Login上启用2FA。

```
network_interface {
  ...
  access_config {}
}
```

This `access_config` block must be set for the proxy to get a public IP, even if the block is empty. The public IP will be ephemeral. If you want, you can assign it a static IP.

即使该块为空，也必须为代理设置此`access_config`块以获取公共IP。 公共IP将是短暂的。 如果需要，可以为其分配一个静态IP。

```
service_account {
  ...
  scopes = ["cloud-platform"]
}
```

Here `scopes` refers to OAuth scopes which affect the Google APIs the service account is allowed to access. Google Cloud already has a permission system called IAM. Adding another layer of OAuth permissions on top of that will needlessly complicate things. You can basically disable them by setting it to `"cloud-platform"`, i.e. the maximum possible scope.

这里的`scopes`是指影响服务帐户被允许访问的Google API的OAuth范围。 Google Cloud已经有一个名为IAM的权限系统。 在此之上添加另一层OAuth权限将不必要地使事情复杂化。 您基本上可以通过将其设置为`"cloud-platform"`来禁用它们，即最大可能范围。

```
metadata_startup_script = templatefile("${path.module}/run_cloud_sql_proxy.tpl", { ... })
```

This is the script that the proxy instance will run on startup. The script is just a bash script with some Terraform-style `${}` string interpolation, hence the `.tpl` (template) extension. The string interpolation is important, because this is how we pass the service account key to Cloud SQL Proxy.

这是代理实例将在启动时运行的脚本。 该脚本只是一个bash脚本，带有一些Terraform样式的`${}`字符串插值，因此具有`.tpl` (模板)扩展名。 字符串插值很重要，因为这是我们将服务帐户密钥传递给Cloud SQL Proxy的方式。

```
#!/bin/bash
set -euo pipefailecho '${service_account_key}' >/var/svc_account_key.json
chmod 444 /var/svc_account_key.jsondocker pull gcr.io/cloudsql-docker/gce-proxy:latest
docker run \
  --rm \
  -p 127.0.0.1:5432:3306 \
  -v /var/svc_account_key.json:/key.json:ro \
  gcr.io/cloudsql-docker/gce-proxy:latest /cloud_sql_proxy \
    -credential_file=/key.json
    -ip_address_types=PRIVATE
    -instances=${db_instance_name}=tcp:0.0.0.0:3306
```

`cloud_sql_proxy` will only accept the service account key as a file. It can't be passed through an env var or by any other means. So we start by `echo`-ing the JSON key and saving it to the `/var` directory. `/var` is important because Container-Optimized OS only gives you a couple directories that are writeable and persistent across reboots (I told you it was very secure!). We want the key to be persistent because Google will occasionally reboot the VM for updates.

`cloud_sql_proxy`将仅接受服务帐户密钥作为文件。 它不能通过环境变量或任何其他方式传递。 因此，我们从`echo` -ing JSON密钥并将其保存到`/var`目录开始。 `/var`很重要，因为Container-Optimized OS仅为您提供了几个在重新引导后可写且可持久的目录(我告诉您这是非常安全的！)。 我们希望该密钥具有持久性，因为Google有时会重新引导VM以进行更新。

```
echo '${service_account_key}' >/var/svc_account_key.json
```

Notice that I'm using single quotes around the `'${service_account_key}'`. If you use double quotes, your JSON key will be mangled. We're risking summoning Cthulhu here by interpolating JSON into Terraform into bash, so just trust me about using single quotes. Also, you must pass `base64decode(google_service_account_key.key.private_key)` for the value of `${service_account_key}`, because the key created by Terraform is base64 encoded.

请注意，我在`'${service_account_key}'`周围使用单引号。 如果使用双引号，则将破坏JSON密钥。 我们冒着通过将JSON插入Terraform到bash中召唤Cthulhu的风险，所以请相信我有关使用单引号的信息。 另外，您必须传递`base64decode(google_service_account_key.key.private_key)`作为`${service_account_key}`的值，因为Terraform创建的密钥是base64编码的。

Next, let's look at the `docker` commands.

接下来，让我们看一下`docker`命令。

```
docker pull gcr.io/cloudsql-docker/gce-proxy:latest
```

Every time the server boots, we want it to pull the latest tagged release of Cloud SQL Proxy. If there are any security updates to Cloud SQL Proxy, we only need to reboot the VM.

每次服务器启动时，我们希望它拉出最新的带标记的Cloud SQL Proxy版本。 如果Cloud SQL Proxy有任何安全性更新，我们只需要重新启动VM。

Next, I'll explain all the flags to that crazy `docker run` command.

接下来，我将解释该疯狂的`docker run`命令的所有标志。

```
docker run -p 127.0.0.1:5432:3306 ...
```

`cloud_sql_proxy` serves on port 3306, even when the db is Postgres. I've taken the liberty of mapping 3306 to the Postgres default port 5432. If you're running MySQL, simply change that flag to `-p 127.0.0.1:3306:3306`. Notice that I'm only opening the container port for `127.0.0.1`. That's because I plan to use the proxy by first `ssh`-ing into the proxy box, then running `psql` in a container. I'll show you the `ssh` command at the end that makes this all work. If you need to connect apps to your db, you'll need to change this flag to just `-p 5432:3306`.

即使数据库是Postgres， `cloud_sql_proxy`可以在端口3306上使用。 我已经自由地将3306映射到Postgres的默认端口5432。如果您正在运行MySQL，只需将该标志更改为`-p 127.0.0.1:3306:3306` 。 注意，我只打开`127.0.0.1`的容器端口。 那是因为我打算首先使用代理服务器`ssh` -ing到代理服务器中，然后运行`psql`容器中。 最后，我将向您展示`ssh`命令，使所有这些工作正常进行。 如果需要将应用程序连接到数据库，则需要将此标志更改为just `-p 5432:3306` 。

```
docker run -v /var/svc_account_key.json:/key.json:ro ...
```

This is how we get the service account key file into the proxy container. The `cloud_sql_proxy` binary in the container will look for the file at `/key.json`.

这就是我们将服务帐户密钥文件放入代理容器的方式。 容器中的`cloud_sql_proxy`二进制文件将在`/key.json`查找文件。

```
-ip_address_types=PRIVATE
```

This is a `cloud_sql_proxy` flag that forces it to only connect to the db using a private IP address. The proxy will encrypt all the traffic it sends to this address.

这是一个`cloud_sql_proxy`标志，强制其仅使用私有IP地址连接到数据库。 代理将加密发送到该地址的所有流量。

```
-instances=${db_instance_name}=tcp:0.0.0.0:3306
```

Finally, we specify the db instance we want to connect to (e.g. `my-project:us-central1:my-instance`). We also specify that we want `cloud_sql_proxy` to serve TCP on port 3306 (by default it uses Unix domain sockets).

最后，我们指定要连接的数据库实例(例如`my-project:us-central1:my-instance`)。 我们还指定我们希望`cloud_sql_proxy`在端口3306上提供TCP服务(默认情况下，它使用Unix域套接字)。

If you've made it this far, we've finally got a secure proxy connected to our private db. There's just one more thing to do.

如果到目前为止，我们终于将安全代理连接到我们的私有数据库。 还有一件事情要做。

通过SSH连接到数据库 (Connecting to the db through SSH)

We're going to `ssh` into the proxy box and run a `psql` container that connects to the db. We're going to do this with one command.

我们将ssh放入代理框中，并运行一个连接到数据库的psql容器。 我们将使用一个命令来执行此操作。

First, you'll need your SSH username. You can get it by running this `gcloud` command.

首先，您需要您的SSH用户名。 您可以通过运行此gcloud命令来获取它。

```
> gcloud compute os-login describe-profile | grep username
```

Now we'll connect to the db over SSH. If you're running MySQL you'll need to modify this.

现在，我们将通过SSH连接到数据库。 如果您正在运行MySQL，则需要对其进行修改。

```
ssh -t <username>@<proxy-public-ip-address> docker run --rm --network=host -it postgres:11-alpine psql -U p
```

The `--network=host` flag means that `localhost` will be the same for both the `postgres` container and the VM host. This is important because we've got two containers running side-by-side: a `postgres` container running `psql` and a `gce-proxy` container running `cloud_sql_proxy`. `cloud_sql_proxy` is serving on`localhost`, so `psql` needs to connect to the *same* `localhost`. If this is confusing, I'm sorry! Networking with containers *is* confusing. All you really need to understand is that the `--network=host` flag is required.

`--network=host`标志意味着postgres容器和VM主机的localhost将相同。 这很重要，因为我们有两个容器并排运行：一个运行psql的postgres容器和一个运行cloud_sql_proxy gce-proxy容器。 cloud_sql_proxy在localhost上提供服务，因此psql需要连接到*相同的* localhost 。 如果这令人困惑，对不起！ 使用集装箱联网*是*混淆。 您真正需要了解的只是--network=host标志。

If all went well, you should have successfully connected to your db. If you got lost, that's okay. This whole process is bananas complicated. Please check out the GitHub repo for a full working example.

如果一切顺利，则应该已经成功连接到数据库。 如果您迷路了，那没关系。 这整个过程很复杂。 请查看GitHub存储库以获取完整的工作示例。

结论 (Conclusion)

There's more to database security than just network isolation, but if you made it this far you've got a very solid security foundation. All you have to do to maintain this infra is:

数据库安全不仅限于网络隔离，而且，如果到目前为止，您将拥有非常坚实的安全基础。 维护此基础结构所需要做的就是：

1. Keep your private SSH keys private.
   将您的私有SSH密钥设为私有。
2. Restart the proxy instance every once in a while so it can auto-update.
   偶尔重新启动代理实例，以便它可以自动更新。

If you do this and *still* get hacked, it's probably not your fault □

如果您这样做*仍然*遭到黑客入侵，那可能不是您的错□

> 翻译自: https://medium.com/swlh/how-to-deploy-a-cloud-sql-db-with-a-private-ip-only-using-terraform-e184b08eca64

terraform