Google Cloud Platform (GCP) | Bucket Enumeration and Privilege Escalation

By Spencer Gietzen

Published: 2019-02-26 · Archived: 2026-04-05 21:10:06 UTC

Intro: Google Cloud Platform (GCP) Security

For those unfamiliar, GCP is a cloud platform that offers a variety of cloud-computing solutions for businesses of any size to take advantage of. Most people would put it up in the “big 3” cloud providers that are available, those being Amazon Web Services (AWS), Microsoft Azure, and GCP.

Cloud security is an extremely important field of study and it is becoming more and more critical for users of these cloud platforms to understand it and embrace it. GCP security is one area of research that is seemingly untouched compared to its competitor, AWS. This could be for a lot of different reasons, with the main reason likely being the market share that each of them control. According to an article recently released on ZDNet.com, AWS has the largest market share, followed by Microsoft Azure, followed by GCP, so it makes sense that AWS security (and Azure security to a much smaller degree) would be far ahead of where GCP is. This doesn’t mean GCP is “insecure” at all, but more so that there is less external, 3rd-party research into GCP security.

As it is an up-and-comer, Rhino has been researching GCP security behind-the-scenes, with our first official blog post on the topic relating to Google Storage security.

Google Storage / Bucket Security

Google Storage is a service offering through GCP that provides static file hosting within resources known as “buckets”. If you’re familiar with AWS, Google Storage is GCP’s version of AWS Simple Storage Service (S3) and an S3 bucket would be equivalent to a Google Storage bucket across the two clouds.

GCP Bucket Policies and Enumeration

Google Storage buckets permissions policies can get very fine-grained though. By design, they can be exposed to a variety of sources (other accounts, organizations, users, etc) which includes being open to the public internet or all authenticated GCP users.

For this reason, we have decided to release a tool that has been used internally for reconnaissance at Rhino for some time known as GCPBucketBrute.

Another Cloud Bucket Enumerator?

Sure, you could say that, but GCPBucketBrute is bringing something new to the idea of “bucket bruteforcing” in that it is expanded to another cloud outside of AWS. There are countless AWS S3 bucket enumerators out there online, but none (that we could find, at least) that targeted other similar storage services, such as Google Storage.

This tool is necessary because of the lack of multi-cloud research and as time goes on, more and more companies are expanding their environments across multiple cloud providers and companies that never have used the cloud are making the leap.

Another benefit of GCPBucketBrute is that it allows you to check every discovered bucket for what privileges you have on that bucket so you can determine your access and if the bucket is vulnerable to privilege escalation. This is outlined further below.

It provides a thorough and customizable interface to find (and abuse) open/misconfigured Google Storage buckets. Here at Rhino, this tool has proven to be a necessity in our Google Cloud penetration tests, web application penetration tests, and red team engagements.

The Tool: GCPBucketBrute

GCPBucketBrute is currently available on our GitHub: https://github.com/RhinoSecurityLabs/GCPBucketBrute

The tool is written in Python 3 and only requires a few different libraries, so it is simple to install. To do so, follow these steps:

1. git clone https://github.com/RhinoSecurityLabs/GCPBucketBrute.git
2. cd GCPBucketBrute && pip3 install -r requirements.txt

With it installed, you can move down to the next section of this post to get started, or you can run the following command to check out the help output on your own:

python3 gcpbucketbrute.py –help

How it Works

Instead of using the “gsutil” (Google Storage Utility) CLI program to perform the enumeration, it was found to be a lot faster to just hit the HTTP endpoint of each bucket we are looking for to check for existence, because there is no overhead from the “gsutil” CLI involved in that case. It also uses subprocesses instead of threads for concurrent execution by design.

When the script starts, it will generate a list of permutations based on the keyword that you supply to the “-k/–keyword” argument. Then, it will start bruteforcing buckets by sending HTTP requests to the Google APIs and it will determine the existence of a bucket based on the HTTP response code. By making HTTP HEAD requests instead of HTTP GET requests, we can make sure the HTTP response does not contain a body, while still getting valid response codes. Although the difference may be negligible, it is theoretically faster for a smaller response (i.e. a response without a body from a HEAD request) to arrive and be parsed than a bigger response (i.e. a response with a body from a GET request) to arrive and be parsed. Using HEAD requests also allows Google’s servers to work a little bit less when trying to process our requests, which is helpful at a mass scale.

Each HTTP HEAD request will be made to the following URL: “https://www.googleapis.com/storage/v1/b/BUCKET_NAME”, where “BUCKET_NAME” is replaced by the current guess. If the HTTP response code is “404” or “400”, then the bucket does not exist. Based on what was discovered during testing, any other HTTP response code we encountered indicates that the bucket exists.

For any bucket that is discovered, the Google Storage TestIamPermissions API will be used to determine what level of access (if any) we have to the target bucket. If credentials are passed into the script, then the results from both an authenticated TestIamPermissions and an unauthenticated TestIamPermissions call will be output to the screen for comparison, so you can see the difference between access granted to allAuthenticatedUsers and allUsers. If no credentials are passed in, only the unauthenticated check will be made and output (to the screen and the out-file if passed into the “-o/–out-file” argument).

If it is found that the user has any permissions (authenticated or not) to the bucket, then all of the permissions will be output. Prior to this, the tool will check for a few common misconfigurations and will output a separate line to make things a little more clear. For example, a bucket that grants the “storage.objects.list” permission to “allUsers” would output the message “UNAUTHENTICATED LISTABLE (storage.objects.list)” prior to outputting all the permissions. This is just to make it a little more clear when there is a misconfiguration in the target bucket.

The list of permissions is also checked to see if the user has access to escalate their permissions on the bucket by modifying the bucket policy. More on this is written below.

Quickstart Examples

Note: If you don’t pass in any authentication-related arguments, you will be prompted by the script for what you want to do (service account, access token, default credentials, unauthenticated).

Scan for buckets using “netflix” as the keyword, using 5 concurrent subprocesses (default), prompting for the authentication type to check authenticated-list permissions on any found buckets (default):

```
python3 gcpbucketbrute.py -k netflix
```

Scan for buckets using “google” as the keyword, using 10 concurrent subprocesses, while staying unauthenticated:

```
python3 gcpbucketbrute.py -k google -s 10 -u
```

Scan for buckets using “apple” as the keyword using 5 concurrent subprocesses (default), prompting for the authentication type to check authenticated-list permissions on any found buckets (default), and outputting results to “out.txt”:

```
python3 gcpbucketbrute.py -k apple -o out.txt
```

Scan for buckets using “android” as the keyword, using 5 concurrent subprocesses (default), while authenticating with a GCP service account whose credentials are stored in sa.pem:

```
python3 gcpbucketbrute.py -k android -f sa.pem
```

Scan for buckets using “samsung” as the keyword, using 8 concurrent subprocesses, while authenticating with a GCP service account whose credentials are stored in service-account.pem:

```
python3 gcpbucketbrute.py -k samsung -s 8 -f service-account.pem
```

Reviewing GCP Buckets in Alexa Top 10k

Following the trend of our S3 bucket enumeration blog post, we went ahead and used GCPBucketBrute to scan the top 10,000 sites according to Alexa.com’s top sites list. This process entailed grabbing the top 10,000 websites, stripping the top-level domains (TLDs), then running GCPBucketBrute against each of the base domains. For example, something like “netflix.com” was stripped of “.com” and we just used “netflix” as the keyword. GCPBucketBrute automatically will remove any duplicates in its wordlist, then it will remove any that are less than 3 characters in length or greater than 63 characters in length, because that is how Google Storage places restrictions on bucket names.

These were our findings:

- 18,618 total buckets were discovered
- 29 buckets of the total 18,618 (~0.16%) allowed read access to all authenticated GCP users, but not unauthenticated users (allAuthenticatedUsers)
- 715 buckets of the total 18,618 (~3.84%) allowed read access to any user on the web (allUsers)
- The remaining 17,874 (~96%) were locked down

In addition to looking for publicly available buckets, we decided to check every bucket found for privilege escalation as well.

Google Storage Bucket Privilege Escalation

Just like AWS S3 buckets can be vulnerable to privilege escalation through misconfigured bucket ACLs (discussed in-depth here), Google Storage buckets can be vulnerable to the same sort of attack.

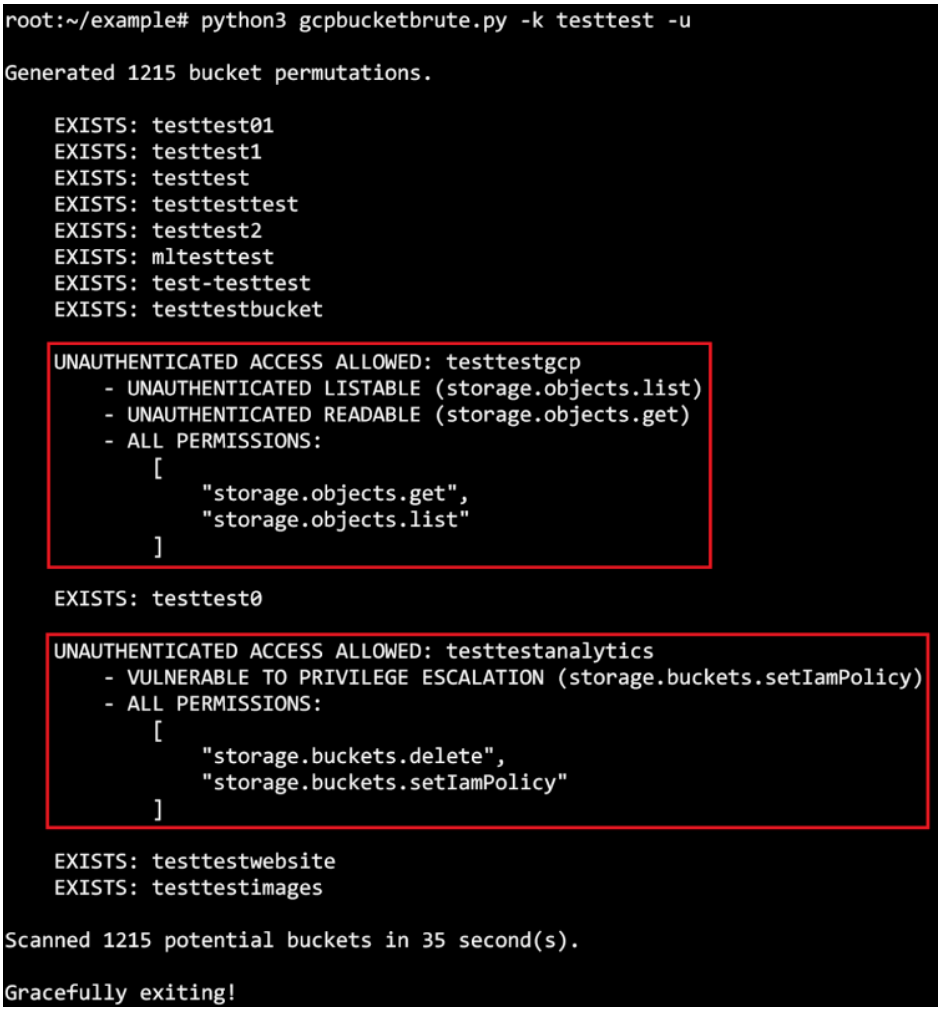
Similar to how GCPBucketBrute checks for open Google Storage buckets through a direct HTTP request to “https://www.googleapis.com/storage/v1/b/BUCKET_NAME/o”, we could check for the bucket’s policy by making direct HTTP requests to “https://www.googleapis.com/storage/v1/b/BUCKET_NAME/iam”, or we can use the “gsutil” CLI tool to run “gsutil iam get gs://BUCKET_NAME”. If “allUsers” or “allAuthenticatedUsers” are allowed to read the bucket policy, we will receive a valid response when pulling the bucket policy, otherwise we will get access denied.

The bucket policy is helpful, but that requires we have the “storage.buckets.getIamPolicy” permission, which we might not have. What if there was a way to determine what permissions we are granted without needing to look at the bucket policy? Wait, there is! The Google Storage “TestIamPermissions” API allows us to supply a bucket name and list of Google Storage permissions, and it will respond with the permissions we (the user making the API request) have on that bucket. This completely bypasses the requirement of viewing the bucket policy and could potentially even give us better information (in the case of a custom role being used).

To determine what permissions we have on a bucket, we can make a request to a URL similar to “https://www.googleapis.com/storage/v1/b/BUCKET_NAME/iam/testPermissions?permissions=storage.objects.list”, where it will respond and let us know if we have the “storage.objects.list” permission on the bucket “BUCKET_NAME”. The permissions parameter can be passed multiple times to check multiple permissions at once, and the Google Storage Python library supports the test_iam_permissions API (even though “gsutil” does not).

GCPBucketBrute will use the current credentials (or none/anonymous if running an unauthenticated scan) to determine what privileges we are granted to every bucket that is discovered. Like mentioned above, for buckets that we have no access to, nothing will output. For buckets that we have *some* access to, it will output a list of what permissions we have. For buckets that we have enough access to privilege escalate and become a full bucket administrator, it will output the permissions and a message indicating it is vulnerable to privilege escalation.

The following screenshot shows what is output when finding a bucket with a few privileges alongside a bucket that is vulnerable to privilege escalation.



For our scan of the Alexa top 10,000, we used the TestIamPermissions API to check what access we were granted and to see if any of the buckets were vulnerable to privilege escalation. For each bucket, that means we made an authenticated request (with our personal GCP credentials) to see what access was granted to “allAuthenticatedUsers”. To also confirm what access was granted to unauthenticated users (allUsers), we ran the same check while unauthenticated. Although it is a very serious misconfiguration to grant all GCP users and/or all unauthenticated users high-level bucket privileges, it turned out to be more common than we thought.

Out of our Alexa top 10,000 scan, we discovered 13 buckets that were vulnerable to privilege escalation to a full bucket admin (~0.07% of the total buckets found) and 21 buckets that were already granting the public internet full bucket admin privileges (~0.11% of the total buckets found).

For the buckets that were reported vulnerable to privilege escalation, this essentially meant that the bucket policy allowed either “allUsers” or “allAuthenticatedUsers” to **write to their bucket policy** (the storage.buckets.setIamPolicy permission). This allowed us to write to the policy that “allUsers”/”allAuthenticatedUsers” were bucket owners, granting us full access to the bucket.

For the buckets that were discovered to be vulnerable to public privilege escalation, we reported the finding to companies that owned them (where we could identify such a thing, the rest were reported directly to Google).

To perform the privilege escalation, we followed these steps:

1. Scanned for existing buckets given a keyword we supplied
2. For any buckets found, check what privileges were granted to “allUsers” or “allAuthenticatedUsers” by using the TestIamPermissions API as both an authenticated and unauthenticated user. If it was found that they had permission to write to the bucket policy (storage.buckets.setIamPolicy), we would have privilege escalation. A vulnerable bucket policy might look something like this:

```
{"bindings":[{"members":["allAuthenticatedUsers","projectEditor:my-test-project","projectOwner:my-test-projec
```

Ignoring most of what is defined in this policy, we can see that the “allAuthenticatedUsers” group is a member of the role “roles/storage.legacyBucketOwner”. If we look at what permissions that role is granted, we see the following:

- storage.buckets.get
- storage.buckets.getIamPolicy
- storage.buckets.setIamPolicy
- storage.buckets.update
- storage.objects.create
- storage.objects.delete
- storage.objects.list

This means that we can read (storage.buckets.getIamPolicy) and write (storage.buckets.setIamPolicy) to the buckets policy and we can create, delete, and list objects within the bucket.

Note that we see the same information by visiting the URL below: (note that the “storage.objects.getIamPolicy” and “storage.objects.setIamPolicy” permissions are omitted because they will throw an error on any bucket that is setup to disable object-level permissions. For buckets that enable object-level permissions, those values can be included).

https://www.googleapis.com/storage/v1/b/BUCKET_NAME/iam/testPermissions?permissions=storage.buckets.delete&permissions=storage.buckets.get&permissions=storage.buckets.getIamPolicy&permissions=storage.buckets.setIamP

If we look at the permissions granted by the Storage Admin role instead, we can see that it grants these privileges:

- firebase.projects.get
- resourcemanager.projects.get
- resourcemanager.projects.list
- storage.buckets.create
- storage.buckets.delete
- storage.buckets.get
- storage.buckets.getIamPolicy
- storage.buckets.list
- storage.buckets.setIamPolicy
- storage.buckets.update
- storage.objects.create
- storage.objects.delete
- storage.objects.get
- storage.objects.getIamPolicy
- storage.objects.list
- storage.objects.setIamPolicy
- storage.objects.update

There are more privileges granted to this role than the role “allAuthenticatedUsers” is a member of, so why don’t we change that?

With the “gsutil” Google Storage CLI program, we can run the following command to grant “allAuthenticatedUsers” access to the “Storage Admin” role, thus escalating the privileges we were granted to the bucket:

gsutil iam ch group:allAuthenticatedUsers:admin gs://BUCKET_NAME

Now if we look at the bucket policy again, we can see the following added to it (because the “ch” command appends instead of overwrites to the policy):

```
{"members":["group:allAuthenticatedUsers"],"role":"roles\/storage.admin"}
```

And just like that, we have escalated our privileges from a Storage Legacy Bucket Owner to a Storage Admin on a bucket that we don’t even own!

One of the main attractions to escalating from a LegacyBucketOwner to Storage Admin is the ability to use the “storage.buckets.delete” privilege. In theory, you could delete the bucket after escalating your privileges, then you could create the bucket in your own account to steal the name.

Now, if we review the privileges we are granted with the TestIamPermissions API again, we see that a few extras are added from the new role we used. Note that not all the privileges allowed by that role will be listed when using the TestIamPermissions API (such as resourcemanager.projects.list) because not all the permissions are Google Storage specific permissions and aren’t supported by the API.

Note: The “gsutil iam ch” command requires permission to read the target bucket’s policy, because it first reads the policy, then adds your addition to it, then writes the new policy. You might not always have this permission, even if you have the SetBucketPolicy permission. In those cases, you would need to overwrite the existing policy and risk causing errors in their environment, such as if you accidentally revoke access from something that needs it.

Disclaimer: Privilege escalation was not actually performed on any of the vulnerable buckets, but instead it was only confirmed the vulnerability existed.

tl;dr:
The Google Storage TestIamPermissions API can be used to determine what level of access we are granted to a specific bucket, regardless of what permissions we actually do have. This allows us to detect when we can write to a buckets policy to grant ourselves a higher level of access to the target bucket.

Conclusion

Even though buckets are created private-by-default, time and time again we see users misconfiguring the permissions on their assets and exposing them to malicious actors in the public and simple APIs like the Google Storage TestIamPermissions just make it easier.

GCPBucketBrute is available right now on our GitHub: https://github.com/RhinoSecurityLabs/GCPBucketBrute

Here at Rhino Security Labs, we perform Google Cloud Platform penetration tests to detect and report on misconfigurations like these from within your environment, rather than from an external perspective. If you want to get started, check out our GCP Penetration Testing Services Page.

For updates and announcements about our research and offerings you can follow us on twitter @RhinoSecurity and you can follow the author of this post/GCPBucketBrute @SpenGietz.
