Backdooring an AWS account

By Daniel Grzelak

Published: 2021-06-08 · Archived: 2026-04-05 22:01:35 UTC

8 min read

Jul 9, 2016

So you've pwned an AWS account — congratulations — now what? You're eager to get to the data theft, *amirite*? Not so fast whipper snapper, have you disrupted logging? Do you know what you have? Sweet! Time to get settled in.

Maintaining persistence in AWS is only limited by your imagination but there are few obvious and oft used techniques everyone should know and watch for.

No one wants to get locked out before mid hack so grab yourself some temporary credentials.

```
aws sts get-session-token --duration-seconds 129600
```

> Acceptable durations for IAM user sessions range from 900 seconds (15 minutes) to 129600 seconds (36 hours), with 43200 seconds (12 hours) as the default. Sessions for AWS account owners are restricted to a maximum of 3600 seconds (one hour). If the duration is longer than one hour, the session for AWS account owners defaults to one hour.

You'll want to setup a cron job to do this regularly from here on out. It might sound crazy, but it ain't no lie. Baby, bye, bye, bye (Sorry got distracted). A sensible person might assume that deleting a compromised access key is a reasonable way to expunge an attacker. Alas, disabling or deleting the original access key does not kill any temporary credentials created with the original. So if you find yourself ousted, you may still get somewhere between 0 and 36 hours to recover.

There are some limitations:

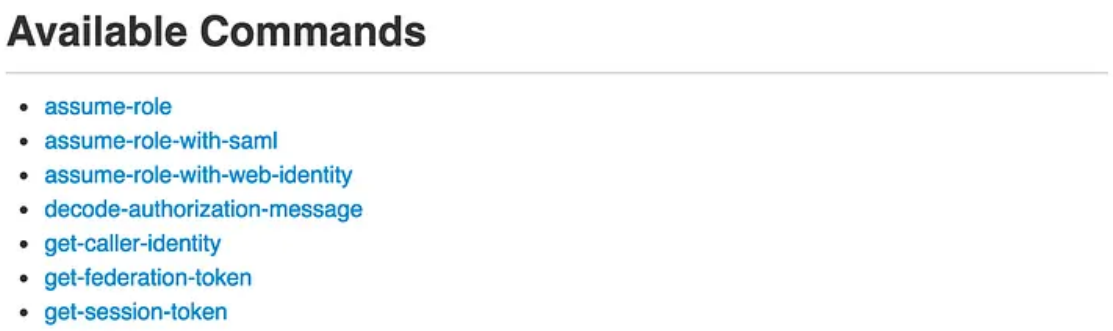
- You cannot call any IAM APIs unless MFA authentication information is included in the request.
- You cannot call any STS API except assume-role.

That does create an annoyance but an annoyance that's trivially overcome. Assuming another role is an API call away. Spinning up compute running under another execution role or instance profile, that can call IAM, is almost as easy.

The best (worst?) part however, is that temporary session keys don't show up anywhere. Checking the web interface or running "aws iam list-access-keys" is ineffective. There's no "list-session-tokens" or "delete-session-token" to go along with "get-session-token". There have been more sitings of the Loch Ness Monster in the wild than AWS session tokens.

This is the entire STS API at time of writing.

Press enter or click to view image in full size



I really do hope Amazon does something about this soon. Having someone use *the force* instead of *the API* within the accounts I'm responsible for genuinely scares me.

Now that you have insurance, it's time to burrow in. If being loud and drunk is your cup of Malört, you could just create a new user and access key. Make it look like an existing user, kind of like typo-squatting, and you'll have yourself a genuine lying-dormant cyber pathogen.

Busting out a new user and key takes two one-liners. Some might call it a two-liner but I'm not into that kind of thing.

```
aws iam create-user --user-name [my-user]
aws iam create-access-key --user-name [my-user]
```

In response, you'll receive an *access key ID* and a *secret access key*, which you'll want to take note of.

```
{
    "AccessKey": {
        "UserName": "[my-user]",
        "Status": "Active",
        "SecretAccessKey": "hunter2",
        "AccessKeyId": "ABCDEFGHIJKLMNOPQRST"
    }
}
```

That approach is nice but it's not the kind of persistent persistence you want. Should the user or access key get discovered, it will take half the API calls to kill them that it did to create them. You'll be left with only stories

about how you used to hack things when you were young. I'll be waiting for you there with my cup of washed-up sadness.

Instead of creating a new account, it's more effective to create a new access key for every user in bulk. Bonus points to those who acquire temporary session tokens at the same time.

The code to do it is straightforward. Even a manager (like me) can write it.

The error handling is somewhat important here as the default key limit per user is two and you will bump up against it semi regularly. Additionally, all access keys have visible creation timestamps which make them easy to spot during a review. Another limitation is that federated (SAML authenticated) users won't be affected as they integrate with roles rather than user accounts.

At this point any good auditor would claim that this was merely a point-in-time activity, leaving potentially risky compliance gaps when new accounts are created in the future. Alas feisty auditors, there is a solution!

Just create a Lambda function that reacts to user creations via a CloudWatch Event Rule and automagically adds a disaster recovery access key and posts it to a PCI-DSS compliant location of your choosing.

AWS Lambda is a server-less compute thingy (only precise technical terms allowed) that runs a function immediately in response to events and automatically manages the underlying infrastructure. CloudWatch Event Rules are a mechanism for notifying other AWS services of state changes in resources in near real time. They have a very natural relationship as CloudWatch provides the sub-system for monitoring AWS API calls and invoking Lambda functions that execute self-contained business logic.

The API calls and deployment packaging required to setup a Lambda function are a bit convoluted but well documented. You can plough through manually and gain valuable plough experience or use a framework like Serverless to avoid unnecessary wear on your delicate hands. Just ensure the function's execute role has the "iam:CreateAccessKey" permission.

Users are so 90s though! Like the Backstreet Boys. Not like Michael Bolten. He's timeless. I mean, how am I supposed to live without him? Now that I've been lovin' him so long.

The AWS recommended ISO* compliant method for escalating privileges is to use the STS *assume role* API call. Amazon describes it so perfectly, I would be robbing you by not quoting it directly.

> For cross-account access, imagine that you own multiple accounts and need to access resources in each account. You could create long-term credentials in each account to access those resources. However, managing all those credentials and remembering which one can access which account can be time consuming. Instead, you can create one set of long-term credentials in one account and then use temporary security credentials to access all the other accounts by assuming roles in those accounts.

Sold! First, create the role.

```
aws iam create-role \
 --role-name [my-role] \
 --assume-role-policy-document [file://assume-role-policy.json]
```

The assume role policy document must include the ARN of the users, roles or accounts that will be accessing the backdoored role. It's best to specify "[account-id]:root", which acts as a wild card for all users and roles in a given account.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::[account-id]:root"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Then attach a policy to the backdoored role describing the actions it can perform. All of them, IMHO. The pre-canned "AdministratorAccess" policy works a treat as it is analogous to root.

```
aws iam attach-role-policy \
 --policy-arn arn:aws:iam::aws:policy/AdministratorAccess \
 --role-name [my-role]
```

There you have it, a freshly minted role to assume from your other pwned accounts without the hassle of all of managing those pesky extra credentials.

While elegant, this approach does have its disadvantages. At some point in the chain of role assumptions, access credentials are required. In the event those credentials or pwned accounts are discovered and purged, your access will die with them.

As before, it's more effective to backdoor the existing roles in an account than create new ones. The code is trickier this time because it requires massaging of existing assume role policies and their structural edge cases. I've tried to comment them fully in the code below but edge cases may have been missed.

While adding adding access keys to a user leaves a trail of recent creation timestamps, by default there is no easy way to identify which part of a policy has been modified. Defenders may be able to identify that a policy has been

changed, but without external record keeping of previous policy versions, they will be left to comb through each policy to look for bad account trusts. This is made more difficult through randomisation of source account ARNs.

Finally, to future proof it all, create a Lambda function that responds to role creations via a CloudWatch Event Rule. As with the access key example, the below code posts the backdoored ARN to a location of your choosing. You may also want to send the role's permissions and source ARN.

If you were less lazy than me, you could make the code react to UpdateAssumeRolePolicy calls and reintroduce backdoors that are removed.

Sometimes you'll want to maintain access to live resources rather than the AWS API. For those situations there's one other basic access persistence tactic worth discussing in an introductory piece, security groups. Security groups tend to get in the way of such things; SSH and database ports aren't typically accessible to the Internet.

> A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. When you launch an instance in a VPC, you can assign the instance to up to five security groups. Security groups act at the instance level, not the subnet level. Therefore, each instance in a subnet in your VPC could be assigned to a different set of security groups.

In practice, "instances" is broader than just EC2. Security groups could be applied to Lambda functions, RDS databases, and other resources that support VPCs.

By now you know the drill. Creating a new security group or rule and applying it to one or two resources is okay but let's skip that step and just do all of them. Shockingly (can I be shocked by own set definitions?), "all of them" includes the *default* security group. This is important because if a resource does not have a security group associated with it, the *default* security group is implicitly associated.

Some older accounts still have services running "EC2 Classic" mode, which means that modifying only EC2 security groups is not sufficient. Back in the day RDS, ElastiCache, and Redshift had their own implementations of security groups. Their relevant authorise functions would need to be called to get full security group coverage:

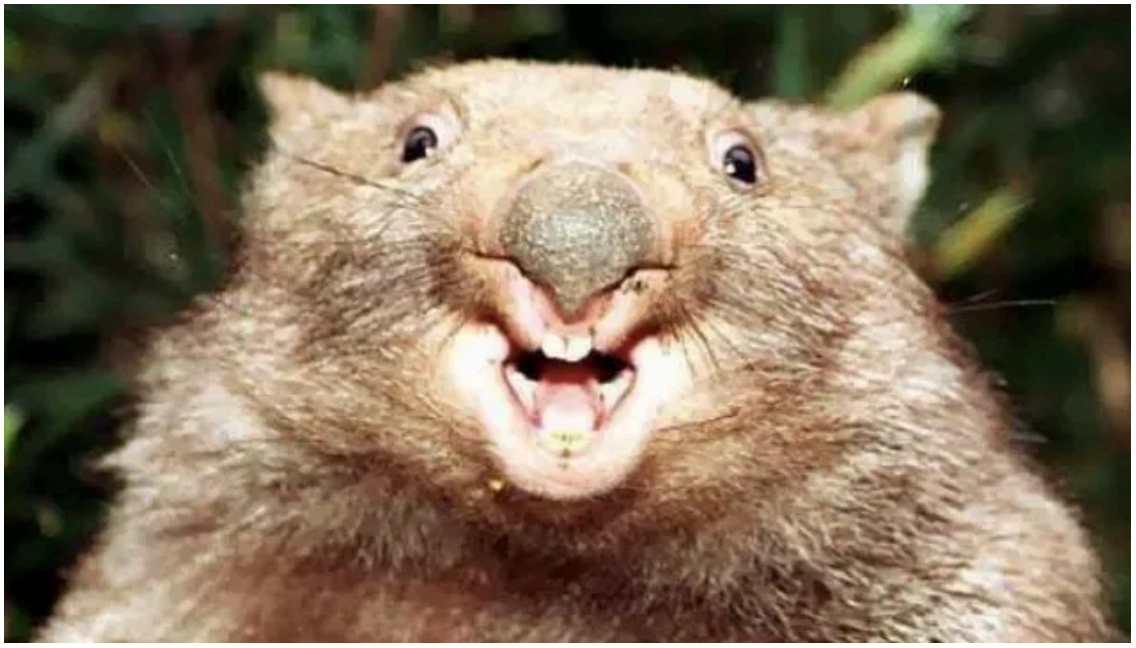
- authorize_db_security_group_ingress
- authorize_cache_security_group_ingress
- authorize_cluster_security_group_ingress

This approach has been phased out. In fact, accounts created after 4th December 2013 cannot use EC2 Classic at all.

Finally, complete the circle of life with a Lambda function that executes when create security group CloudWatch Event Rules are fired.

The extra access rules are pretty easy to spot just by eyeballing the security group. However, the workflow for creating a security group via the web console involves defining all the rules prior to actually calling the API. Consequently, unless someone returns to refine a security group, they are unlikely to notice the extra line item.

Between this and the other tactics, you should be well untruly entrenched in a pwned AWS account. You might not be a devil worm but you are certainly a wombat. An AWS WOMBAT!



It is obvious that information could be used for good and evil. I used it to strengthen the security posture of accounts I am responsible for and make detection processes testable. Professional penetration testers will use it to mimic real world attackers in their engagements. Please do the same. Don't be evil.

Whatever your choice, none of this is unattainable to even the scriptiest (anyone know why there's a red underline under that word? hmmm) of script kiddies. It's better for everyone to have access to the knowledge then just the bad guys.

—

Want to learn to hack AWS? I offer immersive online and in-person training to corporate teams at hackaws.cloud

---