Implementing MFA for AWS

Author: Laura Stone

Claps: 203

Date: Jul 12, 2019

One critical requirement of our efforts to enforce security best practices at Klaviyo is implementing Multi-Factor Authentication (MFA) across the organization (GitHub, G Suite, AWS, etc.) as well as including this as a feature of the Klaviyo product itself. This post focuses on how we used Terraform, Python, and Bash to enforce MFA across multiple AWS accounts, and in the process, centralized our IAM user management and access controls.

Since Klaviyo Engineering is a growing team with a strong culture of infrastructure ownership, many engineers and other technical staff manage our secure platform. This means that we must balance the requirements of a secure and resilient infrastructure with enabling the broader engineering team to self-sufficiently manage and operate their services within AWS. Additionally, internal users needed to be able to access and utilize AWS credentials in tandem with other tooling, such as Terraform, kubectl, and boto3. When we began this project, our teammates had user accounts in multiple AWS accounts. Each user was assigned some unique combination of IAM management policies, roles and groups. Many of these IAM resources had grown organically over the years, never been formally audited, and were entirely undocumented.

We began with an audit of AWS usage. How were developers using AWS? What services did they need access to and via what medium (e.g. the CLI vs. the console)? After surveying and communicating with each individual team, we were able to map their needs against current user permissions (IAM policies, roles, and groups).

We also audited how many users across our AWS environments already had enabled MFA devices. Being security-conscious individuals, several engineers already had MFA enabled. However, we created a small script, run as a cronjob in Kubernetes, to periodically poll each AWS account and check MFA adoption over time:

Script to audit MFA usage on AWS

Our version of this script also tied into our metrics platform (StatsD + Graphite + Grafana) so we could check adoption as we rolled out MFA enforcement (more on that below).

Setting Up MFA Enforcement

Once we had an idea of what permissions our users needed, we researched what permissions were needed to enforce MFA on AWS via IAM. Two resources were useful here, the AWS policy example for allowing users to manage their own security credentials and the docs on how to use MFA in AWS.

Combining these two bits of knowledge, we created a Terraform module that outlined the following:

- An IAM policy that would allow users to manage their MFA devices, but nothing else, until they enabled MFA

- Two IAM groups, one for product developers and one for AWS administrators, each of which provides console access and enables members of the group to assume IAM roles corresponding to their group
- Two IAM roles, one for product developers and one for AWS administrators, each of which provides CLI access

Additionally, several supporting IAM policies were created to do things like deny database deletions and allow groups to assume specific roles if MFA was enabled.

Letâ€™s take a look at an example to illustrate what I mean:

First, we created our developer group and role:

```
resource "aws_iam_group" "mfa_developer" {
  name = "DeveloperMFAGroup-${var.environment}"
  path = "/"
}resource "aws_iam_role" "mfa_developer" {
  name = "DeveloperMFARole-${var.environment}"
  path = "/"
  assume_role_policy = "${data.aws_iam_policy_document.allow_assume_role_i
  max_session_duration = "${var.max_session_duration}"
}
```

Note that the above role only allows the AWS assumeRole action to occur if MFA is enabled:

```
data "aws_iam_policy_document" "allow_assume_role_if_mfa" {
  statement {
    sid = "AllowAssumeRoleIfMFAIsPresent"
    actions = [
      "sts:AssumeRole"
    ]
    principals {
      identifiers = ["*"]
      type = "AWS"
    }
    effect = "Allow"
    condition {
      test = "BoolIfExists"
      variable = "aws:MultiFactorAuthPresent"
      values = [
        "true",
      ]
    }
  }
}
```

Now, we can first fix up permissions for the developer role (in this example, we gave developers PowerUserAccess):

```
resource "aws_iam_role_policy_attachment" "developer_mfa_role_power_user_a
  role = "${aws_iam_role.mfa_developer.name}"
  policy_arn = "arn:aws:iam::aws:policy/PowerUserAccess"
}
```

Because weâ€™re using Terraform, we used booleans to determine whether or not additional policies should be applied. For example, we could conditionally allow developers access to IAM (e.g. in a dev/test environment) or conditionally deny developers from deleting RDS databases:

```
resource "aws_iam_role_policy_attachment" "developer_role_allow_iam_full_a
  count = "${var.allow_iam_full_access ? 1 : 0}"
  role = "${aws_iam_role.mfa_developer.name}"
  policy_arn = "arn:aws:iam::aws:policy/IAMFullAccess"
}resource "aws_iam_role_policy_attachment" "developer_mfa_role_deny_db_del
  count = "${var.deny_db_deletion? 1: 0}"
  role = "${aws_iam_role.mfa_developer.name}"
  policy_arn = "${aws_iam_policy.deny_rds_db_delete.arn}"
}
```

Last, we allowed our DeveloperMFAGroup to assumeRole to the DeveloperMFARole, as well as lock it down to require MFA and provide read only access to resources in the console:

```
resource "aws_iam_group_policy_attachment" "force_mfa_developer_group" {
  group = "${aws_iam_group.mfa_developer.name}"
  policy_arn = "${aws_iam_policy.force_mfa.arn}"
}resource "aws_iam_group_policy_attachment" "allow_assume_developer_role_d
  group = "${aws_iam_group.mfa_developer.name}"
  policy_arn = "${aws_iam_policy.allow_assume_developer_role_if_mfa.arn}"
}resource "aws_iam_group_policy_attachment" "ec2_access_developer_group" {
  group = "${aws_iam_group.mfa_developer.name}"
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ReadOnlyAccess"
}
```

After applying these changes to our AWS environments, we had the user permissions we wanted to be able to give to engineers.

Centralizing IAM User Management and Access Control

We were now managing all of a userâ€™s permissions in Terraform. We thought to ourselves, â€œwhy not also manage users themselves via Terraform?â€�. To make this change, we had to do two things:

1. Import all existing users into Terraform
2. Create a Terraform module to standardize new user creation

The import was fairly straightforward, we needed to:

1. Create a new main.tf that will act as the source of truth for users
2. Add backend and provider blocks to the file so a statefile can be created

```
terraform {
  backend "s3" {
    bucket = "an-example-bucket"
    key    = "path/to/user-mgmt"
  }
}provider "aws" {
}
```

After creating the file, we wrote a small script to:

1. Get a list of all users in an AWS account
2. Initialize a new Terraform statefile in S3 (where user state will live)
3. Import the user into Terraform

Script to import AWS IAM users into Terraform

Next, we had to standardize new user creation. We created another module for this, which handles all of the components required for onboarding a new IAM user. We settled on a modified version of the terraform examples IAM user module, which adds a user, can give console and/or CLI access, and optionally adds SSH keys. Notably, we require each user to create a PGP key that is used to encrypt the access and secret keys generated by Terraform. We do this to ensure that this sensitive data is not stored in plain text in s3.

We had a great first use-case for this new module. We noticed that AWS does not currently support CLI MFA using YubiKey, but many engineers at Klaviyo use YubiKeys, and we wanted to continue to enable this option as much as possible. So, we worked around this issue by encouraging engineers to use a separate AWS user with TOTP for CLI access, while maintaining their original user for console access via YubiKey.

MFA Enforcement

Now, we were managing all users via Terraform, but had not yet placed our staff into the groups we had created to enforce MFA. The implementation of this was easy, when we wanted to enforce MFA for a particular engineer, we simply moved forward with adding new group permissions to that user:

```
resource "aws_iam_user_group_membership" "test_user_group" {
  user = "${aws_iam_user.test_user.name}"groups = [
    "DeveloperMFAGroup",
  ]
}
```

While we found this more repetitive than using count, automation made this change fairly easy and ultimately will be less work down the line as we add and remove users (as opposed to attempting to remove parts of a list, which is what would occur with count).

However, we did not cut everyone over at once. We started by dogfooding our new policies on the Site Reliability Engineering (SRE) team. Then, we convinced a small number of product engineers to pilot using the new groups. After getting feedback from them and tweaking their permissions, we let the rest of the organization know we would be cutting them over.

In order for the cutover to go smoothly, we created documentation for engineers to follow walking them through the process of setting up their local machines for MFA enablement. In order to achieve this, each user must:

1. Create and enable an MFA device
2. Edit their AWS credentials file (~/.aws/credentials on UNIX systems)
3. Create or add to their AWS config file (~/.aws/config on UNIX systems)

For more details, check out the AWS CLI Profiles section of this blog post: https://blog.jayway.com/2017/11/22/aws-cli-mfa/

The Developer Experience

When rolling out this change, we first enforced MFA for ourselves (within the SRE team). It turns out, having to enter a TOTP token every time you want to use the AWS CLI gets kind of annoying! So, we found a tool that helps securely manage and store AWS credentials: aws-vault. This tool manages AWS sessions and also helped manage the usage of all of our automation tooling. We also made small changes to existing automation tooling to utilize aws-vault or appropriately handle sessions using MFA via boto3.

Lastly, we found several helper tools to complement the usage of aws-vault. These include a custom Bash wrapper for Linux Bash users since 1Password is not available on Linux as an agent, Bash functions to help get credentials from 1Password for OSX Users, and the oh-my-zsh aws-vault plugin for ZSH users.

The Bash wrapper can be added to your ~/.bashrc:

Bash helpers when using aws-vault to manage AWS credentials

Similarly, 1Password AWS MFA functions grab signin information from 1Password and use it to run aws-vault:

```
avprod() {
  # If you are not authenticated for 1pass then authenticate first
  if ! op list items; then
    eval $(op signin ${AWS_ACCOUNT})
  fi
  aws-vault exec --mfa-token="$(op get totp aws-prod)" prod -- zsh
}avdev() {
  # If you are not authenticated for 1pass then authenticate first
  if ! op list items; then
    eval $(op signin ${AWS_ACCOUNT)
  fi
  aws-vault exec --mfa-token="$(op get totp aws-dev)" dev -- zsh
}
```

Lastly, the aws-vault oh-my-zsh plugin is simple and shortens the amount of typing required to use aws-vault.

Lessons Learned

There were several important learnings that came out of this project.

First, managing IAM user management and access control via Terraform seriously upped our security game on this front. Now, we can ensure that all changes have an audit trail (in GitHub) and Terraform being checked into source control follows change management best practices.

Second, focusing on developer education smoothed over the transition for developers. Letting users know they could enable MFA far before they were forced to helped people adopt the workflow change at their own pace. Due to the MFA rollout being messaged as part of a larger push toward security across the organization, many engineers immediately saw the benefit.

Third, a slow rollout was a good approach to making this change. Dogfooding the experience was a great empathy-builder and drove us to find helper tooling that had the added benefit of another layer of security. Soliciting feedback from a pilot group was also a great way to ensure the developer experience stayed as good as possible. We caught issues early, and it enabled us to get more one-on-one time with developers to walk them through the process. They could then pay this knowledge forward and help us when developers being onboarded later had questions.

Lastly, having a variety of access needs was another strong developer experience feature. We support both TOTP and YubiKey-based MFA as much as possible (with the exception of the AWS CLI, which currently does not support YubiKey-based MFA). This helped developers with a uniform MFA experience (as they are also using MFA for other tools they use at Klaviyo).

That said, if we were to do it again, there are three things I would change:

1. Having a pilot group for the rollout caught some issues, but it didnâ€™t help with team-specific permissions issues. In the future, I would roll this change out by team.
2. Although our access controls are significantly more standardized than they used to be, we are now running into problems that would be solved by more granularity for developer permissions. In the future, we should continue to refactor our groups and roles, perhaps implementing team-based access patterns.
3. It really sucks to need to need to create two users for folks who want to use YubiKeys â€” hopefully AWS implements YubiKey for CLI access soon

If this sounds interesting, weâ€™re hiring!