Lainne Munkombwe

The Compliant Cloud Perimeter (AWS + Terraform)

Executive Summary:

This project demonstrates the transition from manual, error-prone cloud configuration to **Infrastructure as Code (IaC)** with a Security-First mindset, the essence of DevSecOps.

Yours truly has engineered a hardened AWS infrastructure designed to satisfy Governance, Risk, and Compliance (GRC) requirements.

By utilizing **Terraform**, I have automated the enforcement of security controls, ensuring that the environment is auditable, repeatable, and resilient against common misconfigurations.

Project Execution Structure

The implementation is organized into eight phases. Phase 1 (with sub-phases 1.1-1.3) establishes foundational security through IAM hardening and identity management. Phase 2 initializes the Infrastructure as Code environment. Phases 3-4 implement Pillars 1-2 (Network Isolation and Micro-Segmentation). Phases 5-6 deploy the compute layer with Pillar 3 (Least Privilege IAM). Phase 7 activates Pillar 4 (Continuous Monitoring). Phase 8 demonstrates iterative hardening by replacing permissive policies with scoped access controls, then decommissions all infrastructure following NIST 800-88 standards.

NOTE: *Yours truly believes in absolute security. While this document showcases the live configuration of the project, please note that all assets, including the AWS account and IAM identities, will be* ***decommissioned and purged*** *immediately upon project completion. This ensures zero residual risk and adheres to the* ***NIST 800-88*** *standard for media sanitization and asset disposal. There are no keys to a kingdom that no longer exists:)*

Technical Architecture & GRC Alignment

The project is built on four core pillars, each mapping directly to industry standard frameworks like NIST or ISO 27001.

1. Network Isolation (Custom VPC)

What: A tailored Virtual Private Cloud featuring distinct public and private subnets across multiple Availability Zones.

Why: Standard default VPCs are often too open. This architecture ensures that sensitive workloads are isolated from the public internet, reducing the attack surface.

2. Security Groups as Code (Micro-Segmentation)

What: Strict ingress/egress rules defined within Terraform.

Why: We operate on a **Zero Trust** principle. There are no `0.0.0.0/0` (open to the world) rules here; only specific ports and protocols are permitted for identified sources.

3. Identity & Access Management (Least Privilege)

What: Automated creation of IAM Roles and Instance Profiles for EC2, replacing the dangerous practice of using hardcoded access keys.

Why: This aligns with the **Principle of Least Privilege (PoLP)**. The infrastructure only has the permissions it absolutely needs to function, and nothing more.

4. Continuous Monitoring (Audit Trail)

What: Integration of **AWS Config** and **CloudWatch Logging**.

Why: In a SOC or GRC role, visibility is everything. This setup ensures that every change is logged and the environment is constantly monitored for compliance drift.

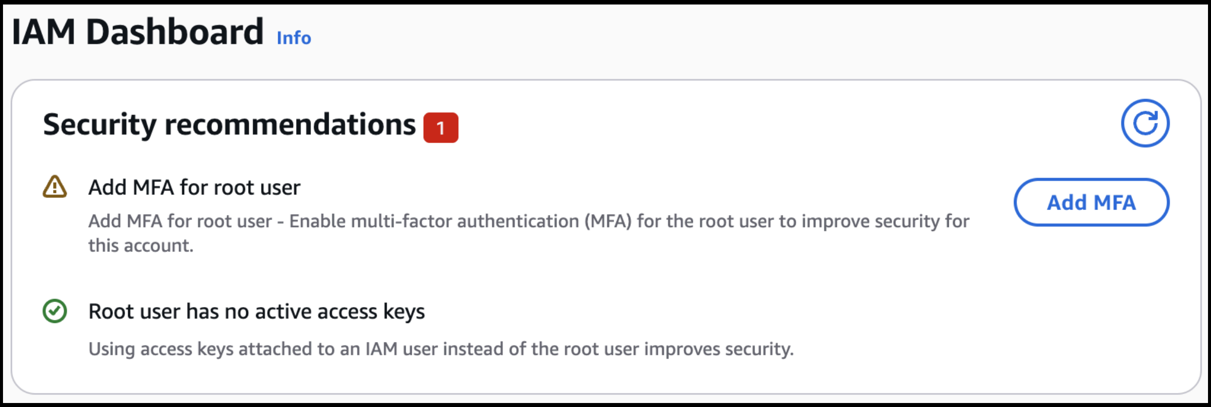
Why This Matters

I recognize that modern security is about finding bugs as well as building systems that are secure by default, this is the essence of DevSecOps.

This project proves my ability to:

- Translate high-level compliance requirements into technical reality.
- Manage cloud resources efficiently using industry-standard tools.
- Maintain a high level of technical rigor.

Phase 1: IAM Hardening


IAM Dashboard Info

Security recommendations 1

Add MFA for root user

Add MFA for root user - Enable multi-factor authentication (MFA) for the root user to improve security for this account.

Root user has no active access keys

Using access keys attached to an IAM user instead of the root user improves security.

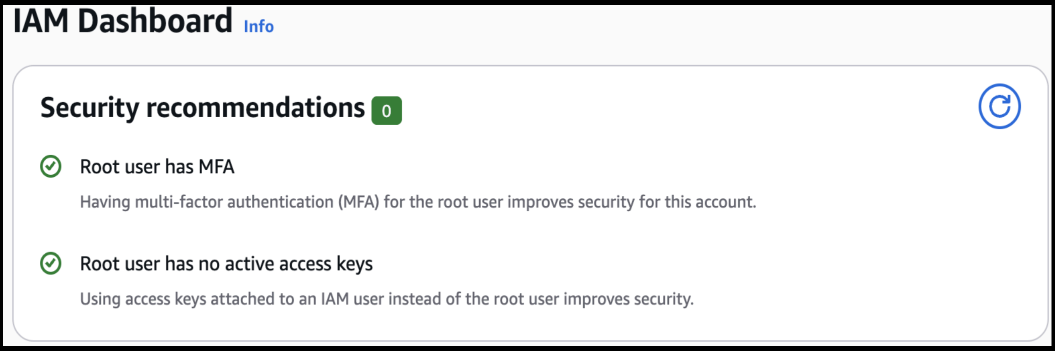
Figure 1: Initial Security Posture Assessment

Result: Vulnerable starting point (no active access keys, no MFA for root user)

I'm going to remedy this by initiating a hardening process, this is to ensure alignment with :

- ACSC Essential Eight

- NIST Cybersecurity Framework



IAM Dashboard Info

Security recommendations 0

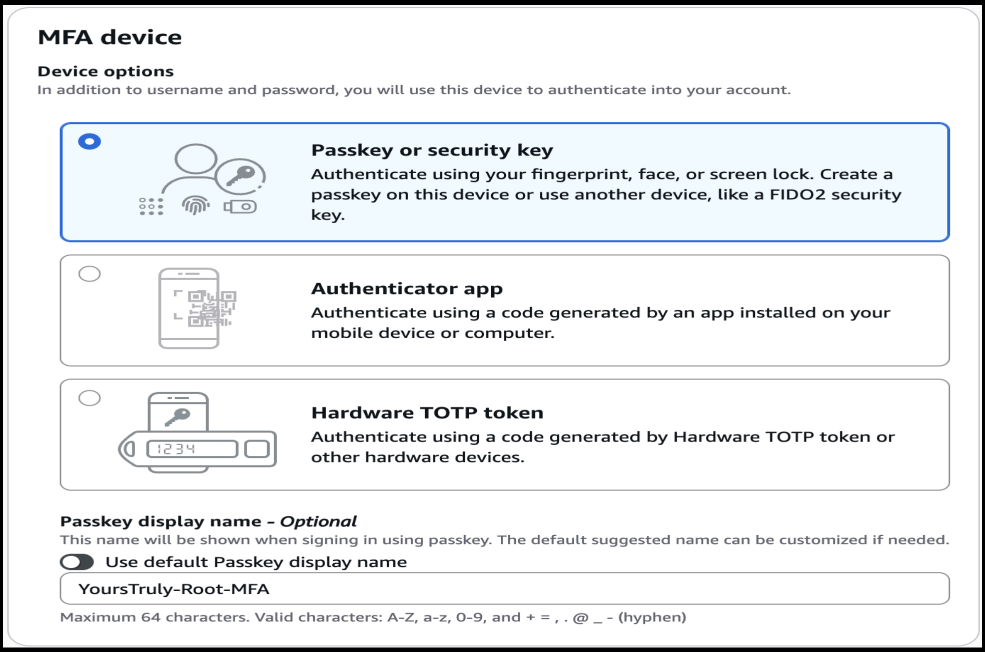
✓ Root user has MFA

Having multi-factor authentication (MFA) for the root user improves security for this account.

✓ Root user has no active access keys

Using access keys attached to an IAM user instead of the root user improves security.

Figure 2: MFA enabled for the root user.



MFA device

Device options

In addition to username and password, you will use this device to authenticate into your account.

● **Passkey or security key** — Authenticate using your fingerprint, face, or screen lock. Create a passkey on this device or use another device, like a FIDO2 security key.

○ **Authenticator app** — Authenticate using a code generated by an app installed on your mobile device or computer.

○ **Hardware TOTP token** — Authenticate using a code generated by Hardware TOTP token or other hardware devices.

Passkey display name - *Optional*

This name will be shown when signing in using passkey. The default suggested name can be customized if needed.

Use default Passkey display name

YoursTruly-Root-MFA

Maximum 64 characters. Valid characters: A-Z, a-z, 0-9, and + = , . @ _ - (hyphen)

Figure 3: Root Account Hardening & MFA Registration

What I'm fixing:

I'm securing the Root user account, which has unrestricted, "god-mode" access to every resource in the AWS environment. An unhardened Root account is a single point of failure and a primary target for credential theft.

How I'm fixing it:

I have enabled Multi-Factor Authentication (MFA) by registering a uniquely named virtual MFA device, YoursTruly-Root-MFA, as the secondary authentication factor.

Why I'm fixing it:

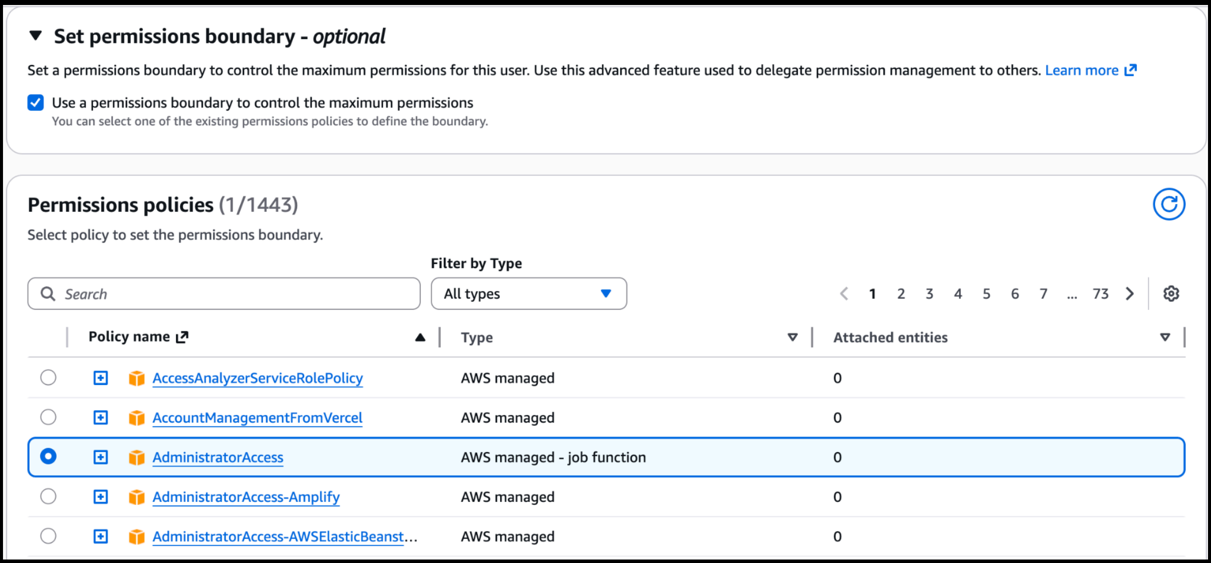
This alignment satisfies NIST CSF PR.AC-1 and the ACSC Essential Eight requirements for protecting privileged accounts. It ensures that even if a password is compromised, an attacker cannot gain access without the physical possession of the MFA device.

Phase 1.1: Establishing the Secure Deployment Identity

Standard Alignment:

ACSC Essential Eight - Restrict administrative privileges.

NIST CSF PR.AC-4 - Managing access permissions (Principal of Least Privilege).



Set permissions boundary - *optional*

Set a permissions boundary to control the maximum permissions for this user. Use this advanced feature used to delegate permission management to others. Learn more

[✓] Use a permissions boundary to control the maximum permissions
You can select one of the existing permissions policies to define the boundary.

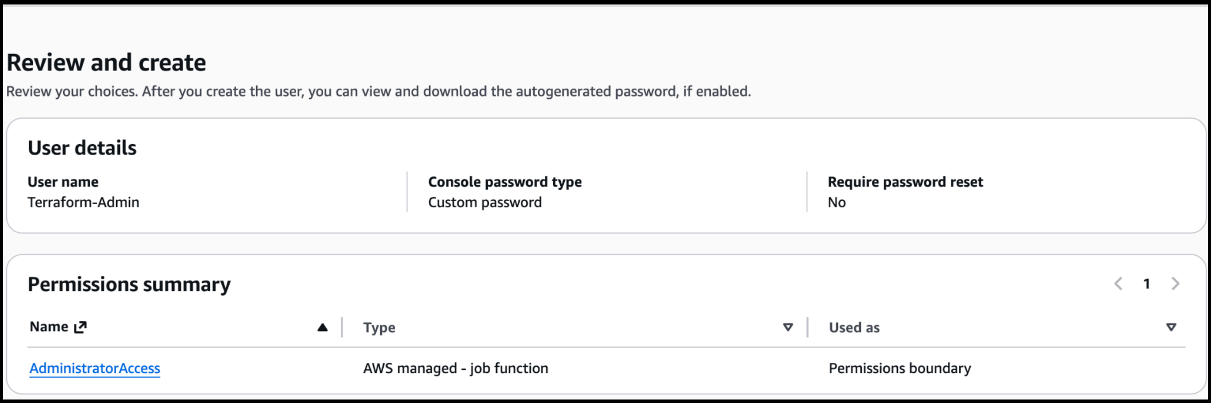
Permissions policies (1/1443)

Select policy to set the permissions boundary.

Filter by Type: All types

	Policy name	Type	Attached entities
○	AccessAnalyzerServiceRolePolicy	AWS managed	0
○	AccountManagementFromVercel	AWS managed	0
●	AdministratorAccess	AWS managed - job function	0
○	AdministratorAccess-Amplify	AWS managed	0
○	AdministratorAccess-AWSElasticBeanst…	AWS managed	0

Figure 4: The AWS IAM Users console confirming the creation of the `Terraform-Admin` service account. This user is the dedicated programmatic identity used exclusively by Terraform, ensuring that all infrastructure changes are tied to a traceable, non-root identity.



Review and create

Review your choices. After you create the user, you can view and download the autogenerated password, if enabled.

User details

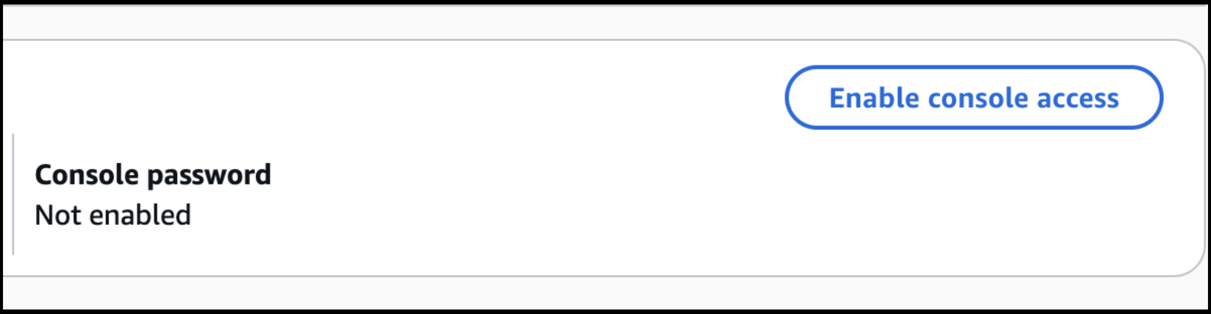
User name	Console password type	Require password reset
Terraform-Admin	Custom password	No

Permissions summary

Name	Type	Used as
AdministratorAccess	AWS managed - job function	Permissions boundary

Figure 5: The `Terraform-Admin` user's access summary page, confirming that AWS

Management Console access has been **disabled**. This enforces the separation between human (console) and machine (programmatic) access, satisfying ACSC Essential Eight requirements for restricting administrative privileges.



Console password
Not enabled

Figure 6:The permissions or group assignment view for the `Terraform-Admin` user, confirming the scope of access granted to this service account is limited strictly to what Terraform requires; no more, no less.

What I'm fixing:

Excessive permissions for a service account.

The security risk of using the Root account for automated infrastructure tasks. In a production environment, using Root for daily tasks is a critical failure of Privileged Access Management (PAM).

How I'm fixing it:

Disabling AWS Management Console access for the `Terraform-Admin` user.

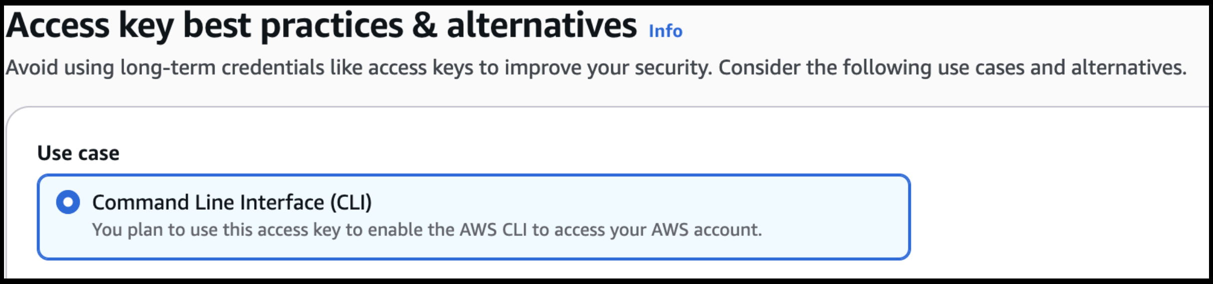
I am creating a dedicated Service Account (Terraform-Admin) specifically for Terraform. This user is restricted to Programmatic Access only, meaning it cannot log into the AWS Web Console.

Why I'm fixing it:

This follows the **Principle of Least Privilege (PoLP)**. This identity is strictly for automation; removing GUI access ensures it cannot be used for manual, non-audited changes in the console, satisfying **ACSC Essential Eight** requirements for restricting administrative access to only what is strictly necessary.

Phase 1.2: Provisioning Programmatic Credentials

Standard Alignment:
NIST CSF PR.AC-4: Managing Access PermissionsACSC Essential Eight: Restrict Administrative Privileges.



Access key best practices & alternatives

Avoid using long-term credentials like access keys to improve your security. Consider the following use cases and alternatives.

Use case

● Command Line Interface (CLI)
You plan to use this access key to enable the AWS CLI to access your AWS account.

Figure 7: "Command Line Interface (CLI)" as the specific access key use case.

What I'm fixing:

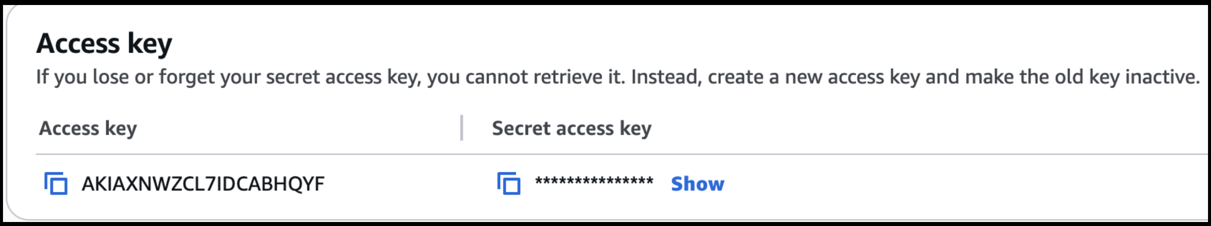
The security risk of using long-term, multi-purpose credentials. By selecting a specific use case, I am ensuring that these keys are designated for their intended function which is automation via the CLI, rather than being generic keys that could be misused across different platforms.

How I'm fixing it:

I am explicitly selecting the Command Line Interface (CLI) use case for the Terraform-Admin user. This triggers AWS to provide specific "best practice" guidance for this credential type, such as storing them in the local .aws/credentials file rather than hardcoding them into scripts.

Why I'm fixing it:

This follows the Principle of Separation of Duties. It ensures that the "machine identity" (the CLI user) is clearly distinguished from a "human identity" (the Console user). From a GRC perspective, this creates a clean audit trail where infrastructure changes are tied specifically to a programmatic intent, making it easier to detect anomalous behavior that falls outside of normal CLI activity.



Access key

If you lose or forget your secret access key, you cannot retrieve it. Instead, create a new access key and make the old key inactive.

Access key	Secret access key
AKIAXNWZCL7IDCABHQYF	*************** Show

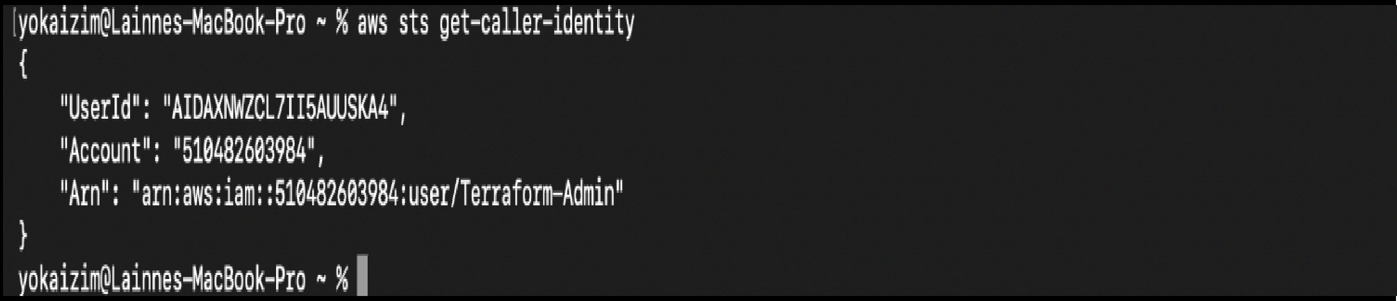
Figure 8: *Finalized the generation of secure API keys. The secret key has been redacted in this report to maintain security integrity, in line with internal data handling policies. This aligns with NIST CSF PR.AC-4 by ensuring that credentials are treated as sensitive assets. By downloading the credentials in a .csv format and securing them locally, I am ensuring that the "Secret" is never stored in the AWS cloud or the source code, reducing the risk of credential exposure.*

Phase 1.3: Validating Programmatic Identity

Standard Alignment:

ACSC Essential Eight: Restrict administrative privileges (Separation of human and machine access).

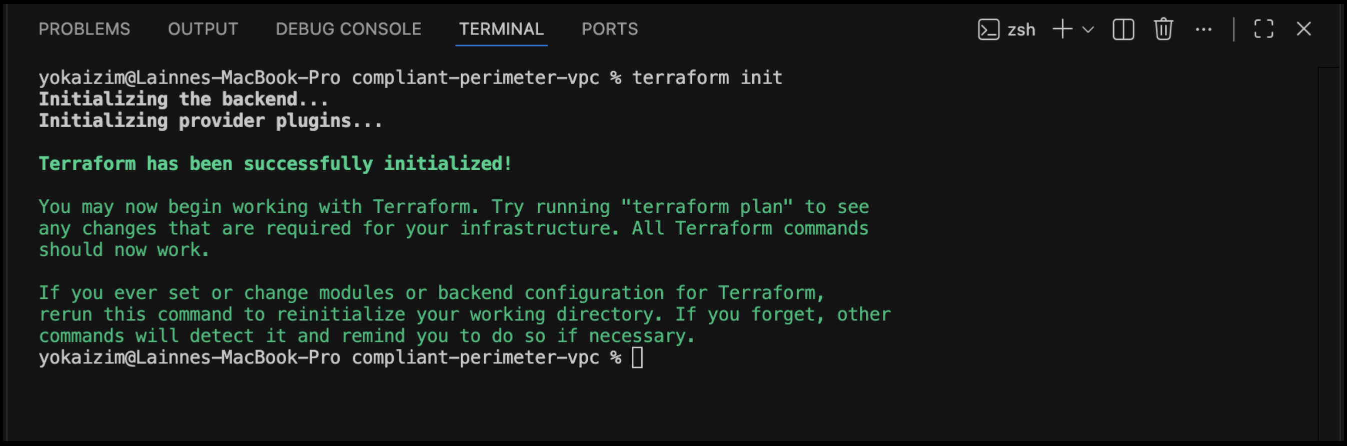
NIST CSF PR.AC-4: Managing access permissions.



```
yokaizim@Lainnes-MacBook-Pro ~ % aws sts get-caller-identity
{
    "UserId": "AIDAXNWZCL7II5AUUSKA4",
    "Account": "510482603984",
    "Arn": "arn:aws:iam::510482603984:user/Terraform-Admin"
}
yokaizim@Lainnes-MacBook-Pro ~ %
```

Figure 9: I am performing a cryptographic "handshake" between my local machine and the AWS Cloud. By executing the get-caller-identity command, I am forcing a real-time verification of the active session. This step satisfies the **Accountability** requirement of **NIST CSF PR.AC-4**. The output (User ID, Account, and ARN) proves that the terminal is correctly authenticated as the restricted `Terraform-Admin` service account. In an audit, this ensures that every subsequent infrastructure change is tied to a specific, non-root identity in the AWS CloudTrail logs, providing a 100% transparent audit trail.

Phase 2: Terraform init



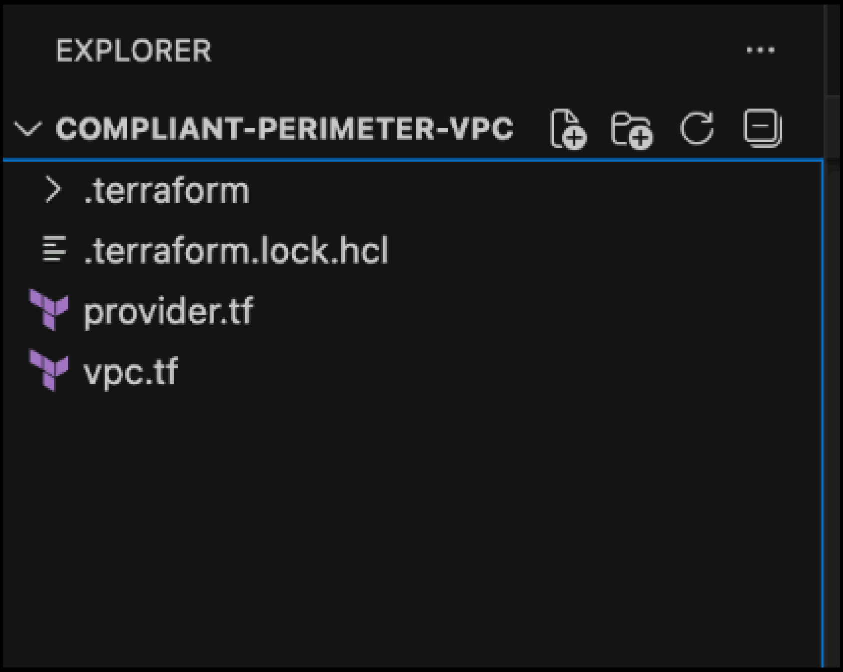
```
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc % terraform init
Initializing the backend...
Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc % 
```

Figure 10: Successfully initialized the Terraform environment.

What I’m fixing:
The dependency on manual, unverified environment setups. Without initialization, the automation engine lacks the specific "drivers" (providers) needed to communicate with the cloud provider safely.

How I’m fixing it:
Executing the terraform init command. This process automatically downloads the authenticated AWS provider plugins and initializes the "backend," ensuring the environment is prepared for secure deployment.

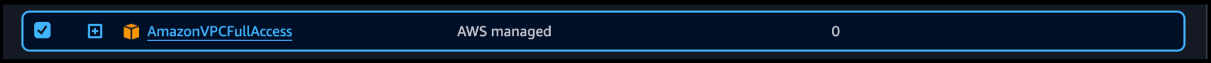
Why I’m fixing it:
This aligns with NIST CSF ID.AM-2 and PR.IP-1 (Baseline Configuration). By initializing the environment through code, I am ensuring a repeatable and consistent deployment process. This eliminates "Configuration Drift" where different environments have different settings, a common cause of security vulnerabilities in the cloud.


EXPLORER

COMPLIANT-PERIMETER-VPC

```
> .terraform
  .terraform.lock.hcl
  provider.tf
  vpc.tf
```

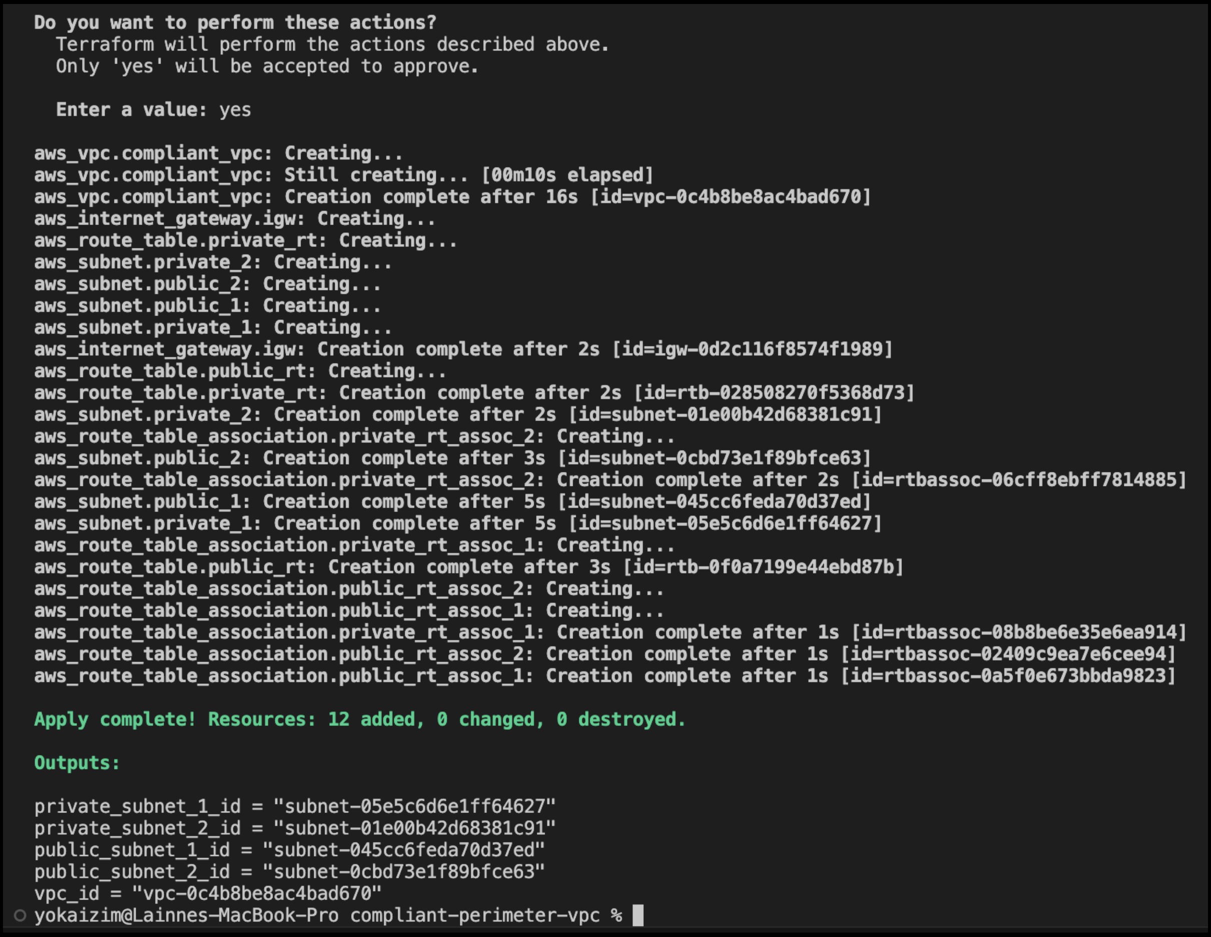
Figure 11:The VS Code editor showing the `provider.tf` file after a successful `terraform init`. The `.terraform` directory and `.terraform.lock.hcl` file are now present in the project, confirming that the AWS provider plugin has been authenticated and downloaded, and that the local environment is locked to a verified provider version.



✓ AmazonVPCFullAccess	AWS managed	0

Figure 12: The Terraform-Admin user initially lacked the permissions required to provision VPC resources, resulting in a 403 Unauthorized error during terraform apply. This is resolved by attaching the AmazonVPCFullAccess managed policy via the AWS IAM Console. This grants the service account the necessary permissions to create and manage VPC, subnet, route table, and internet gateway resources. As noted, this broad policy will be tightened to a custom least-privilege policy in a later phase — demonstrating the iterative hardening approach central to DevSecOps.

Phase 3: Network Isolation (Custom VPC)

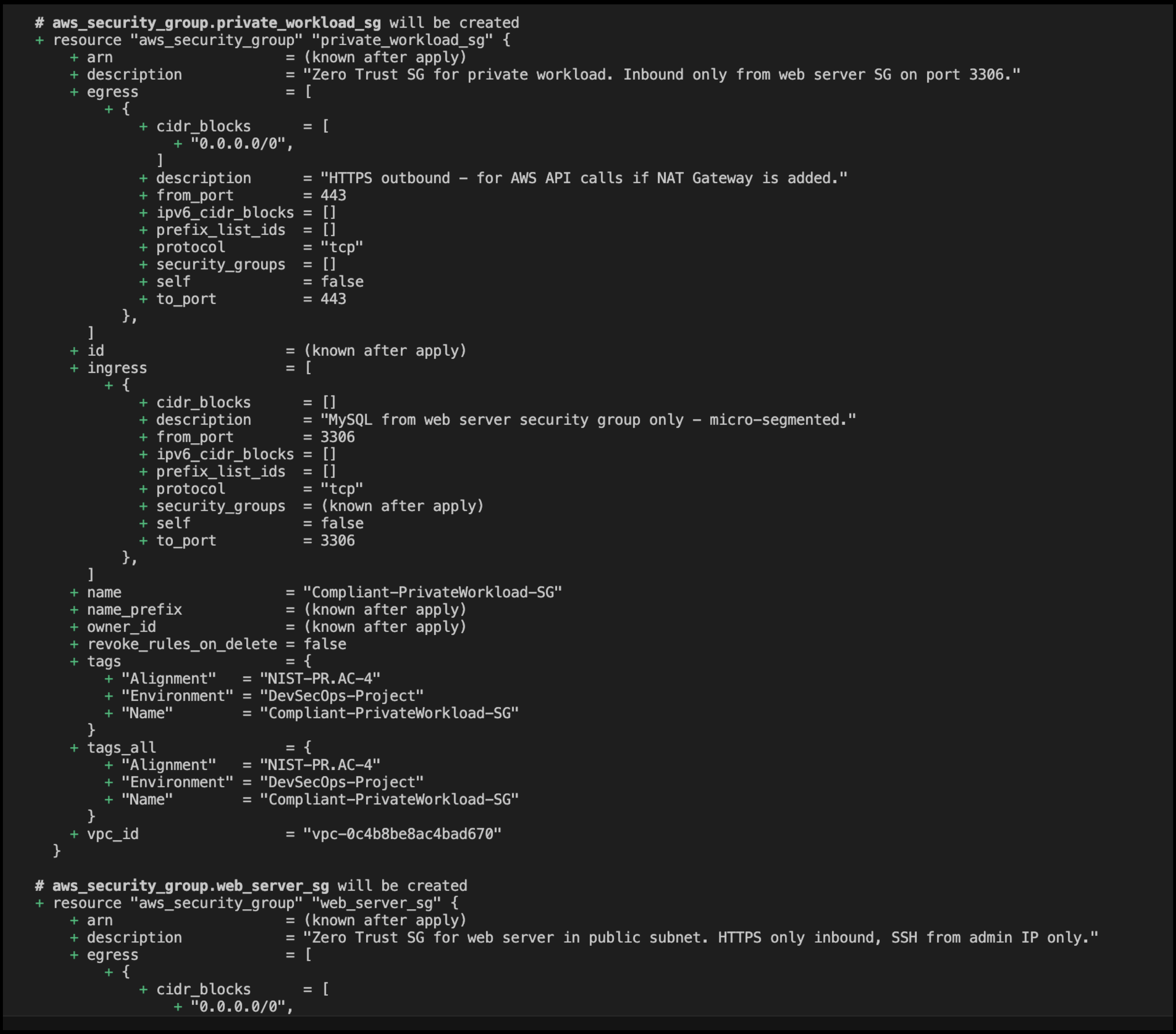


```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_vpc.compliant_vpc: Creating...
aws_vpc.compliant_vpc: Still creating... [00m10s elapsed]
aws_vpc.compliant_vpc: Creation complete after 16s [id=vpc-0c4b8be8ac4bad670]
aws_internet_gateway.igw: Creating...
aws_route_table.private_rt: Creating...
aws_subnet.private_2: Creating...
aws_subnet.public_2: Creating...
aws_subnet.public_1: Creating...
aws_subnet.private_1: Creating...
aws_internet_gateway.igw: Creation complete after 2s [id=igw-0d2c116f8574f1989]
aws_route_table.public_rt: Creating...
aws_route_table.private_rt: Creation complete after 2s [id=rtb-028508270f5368d73]
aws_subnet.private_2: Creation complete after 2s [id=subnet-01e00b42d68381c91]
aws_route_table_association.private_rt_assoc_2: Creating...
aws_subnet.public_2: Creation complete after 3s [id=subnet-0cbd73e1f89bfce63]
aws_route_table_association.private_rt_assoc_2: Creation complete after 2s [id=rtbassoc-06cff8ebff7814885]
aws_subnet.public_1: Creation complete after 5s [id=subnet-045cc6feda70d37ed]
aws_subnet.private_1: Creation complete after 5s [id=subnet-05e5c6d6e1ff64627]
aws_route_table_association.private_rt_assoc_1: Creating...
aws_route_table.public_rt: Creation complete after 3s [id=rtb-0f0a7199e44ebd87b]
aws_route_table_association.public_rt_assoc_2: Creating...
aws_route_table_association.public_rt_assoc_1: Creating...
aws_route_table_association.private_rt_assoc_1: Creation complete after 1s [id=rtbassoc-08b8be6e35e6ea914]
aws_route_table_association.public_rt_assoc_2: Creation complete after 1s [id=rtbassoc-02409c9ea7e6cee94]
aws_route_table_association.public_rt_assoc_1: Creation complete after 1s [id=rtbassoc-0a5f0e673bbda9823]

Apply complete! Resources: 12 added, 0 changed, 0 destroyed.

Outputs:

private_subnet_1_id = "subnet-05e5c6d6e1ff64627"
private_subnet_2_id = "subnet-01e00b42d68381c91"
public_subnet_1_id = "subnet-045cc6feda70d37ed"
public_subnet_2_id = "subnet-0cbd73e1f89bfce63"
vpc_id = "vpc-0c4b8be8ac4bad670"
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc %
```

Figure 13: Successful terraform apply ; VPC infrastructure provisioned.

All 12 resources were created successfully, including the VPC, Internet Gateway, four subnets (two public, two private across two Availability Zones), two route tables, and four route table associations. The output IDs confirm each resource is live in AWS. Critically, the private route table has no route to 0.0.0.0/0, enforcing network-level isolation of private workloads from the public internet , satisfying NIST CSF PR.AC-4 and ISO 27001 A.13.1.

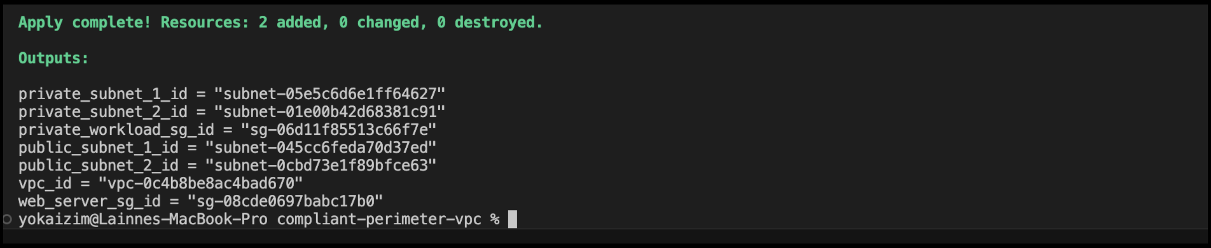
Phase 4: Security Groups (Micro-Segmentation)



```
  # aws_security_group.private_workload_sg will be created
  + resource "aws_security_group" "private_workload_sg" {
      + arn                    = (known after apply)
      + description            = "Zero Trust SG for private workload. Inbound only from web server SG on port 3306."
      + egress                 = [
          + {
              + cidr_blocks      = [
                  + "0.0.0.0/0",
                ]
              + description      = "HTTPS outbound - for AWS API calls if NAT Gateway is added."
              + from_port        = 443
              + ipv6_cidr_blocks = []
              + prefix_list_ids  = []
              + protocol         = "tcp"
              + security_groups  = []
              + self             = false
              + to_port          = 443
            },
        ]
      + id                     = (known after apply)
      + ingress                = [
          + {
              + cidr_blocks      = []
              + description      = "MySQL from web server security group only - micro-segmented."
              + from_port        = 3306
              + ipv6_cidr_blocks = []
              + prefix_list_ids  = []
              + protocol         = "tcp"
              + security_groups  = (known after apply)
              + self             = false
              + to_port          = 3306
            },
        ]
      + name                   = "Compliant-PrivateWorkload-SG"
      + name_prefix            = (known after apply)
      + owner_id               = (known after apply)
      + revoke_rules_on_delete = false
      + tags                   = {
          + "Alignment"   = "NIST-PR.AC-4"
          + "Environment" = "DevSecOps-Project"
          + "Name"        = "Compliant-PrivateWorkload-SG"
        }
      + tags_all               = {
          + "Alignment"   = "NIST-PR.AC-4"
          + "Environment" = "DevSecOps-Project"
          + "Name"        = "Compliant-PrivateWorkload-SG"
        }
      + vpc_id                 = "vpc-0c4b8be8ac4bad670"
    }

  # aws_security_group.web_server_sg will be created
  + resource "aws_security_group" "web_server_sg" {
      + arn                    = (known after apply)
      + description            = "Zero Trust SG for web server in public subnet. HTTPS only inbound, SSH from admin IP only."
      + egress                 = [
          + {
              + cidr_blocks      = [
                  + "0.0.0.0/0",
```

Figure 14: `terraform plan` output - Security Groups

The plan confirms two security groups will be created. The Compliant-WebServer-SG enforces Zero Trust by allowing only HTTPS (443) inbound from the internet and SSH (22) exclusively from the trusted admin IP. The Compliant-PrivateWorkload-SG permits inbound traffic only from the web server's security group on port 3306 (MySQL) - this is micro-segmentation in action, ensuring private workloads are never directly exposed. No open ingress rules exist on any private resource, satisfying NIST CSF PR.AC-4.

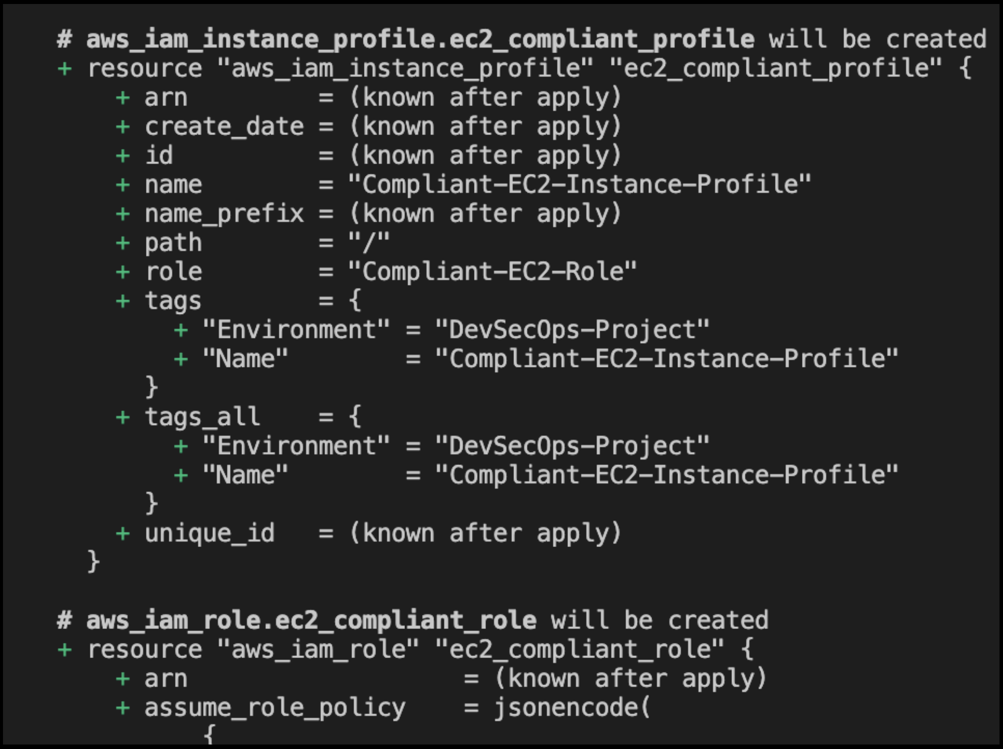


```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

private_subnet_1_id = "subnet-05e5c6d6e1ff64627"
private_subnet_2_id = "subnet-01e00b42d68381c91"
private_workload_sg_id = "sg-06d11f85513c66f7e"
public_subnet_1_id = "subnet-045cc6feda70d37ed"
public_subnet_2_id = "subnet-0cbd73e1f89bfce63"
vpc_id = "vpc-0c4b8be8ac4bad670"
web_server_sg_id = "sg-08cde0697babc17b0"
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc %
```

Figure 15: Successful `terraform apply` - Security Groups provisioned

Both security groups were created successfully and now live in the VPC. The output confirms the `web_server_sg` and `private_workload_sg` IDs alongside the previously provisioned VPC and subnet resources, showing the infrastructure is building up as expected.

Phase 5: IAM Role and Instance Profile



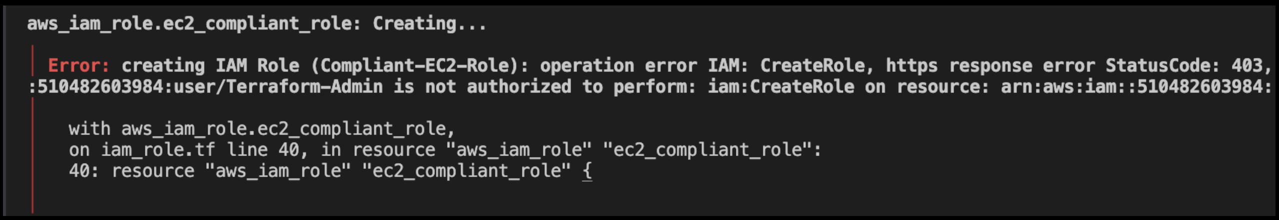
```
  # aws_iam_instance_profile.ec2_compliant_profile will be created
  + resource "aws_iam_instance_profile" "ec2_compliant_profile" {
      + arn         = (known after apply)
      + create_date = (known after apply)
      + id          = (known after apply)
      + name        = "Compliant-EC2-Instance-Profile"
      + name_prefix = (known after apply)
      + path        = "/"
      + role        = "Compliant-EC2-Role"
      + tags        = {
          + "Environment" = "DevSecOps-Project"
          + "Name"        = "Compliant-EC2-Instance-Profile"
        }
      + tags_all    = {
          + "Environment" = "DevSecOps-Project"
          + "Name"        = "Compliant-EC2-Instance-Profile"
        }
      + unique_id   = (known after apply)
    }

  # aws_iam_role.ec2_compliant_role will be created
  + resource "aws_iam_role" "ec2_compliant_role" {
      + arn                   = (known after apply)
      + assume_role_policy    = jsonencode(
            {
```

Figure 16: `terraform plan` output - IAM Role and Instance Profile

What I'm fixing: The dangerous practice of hardcoding access keys directly into EC2 instances. Static, long-lived credentials on an instance are a prime target if the instance is ever compromised.

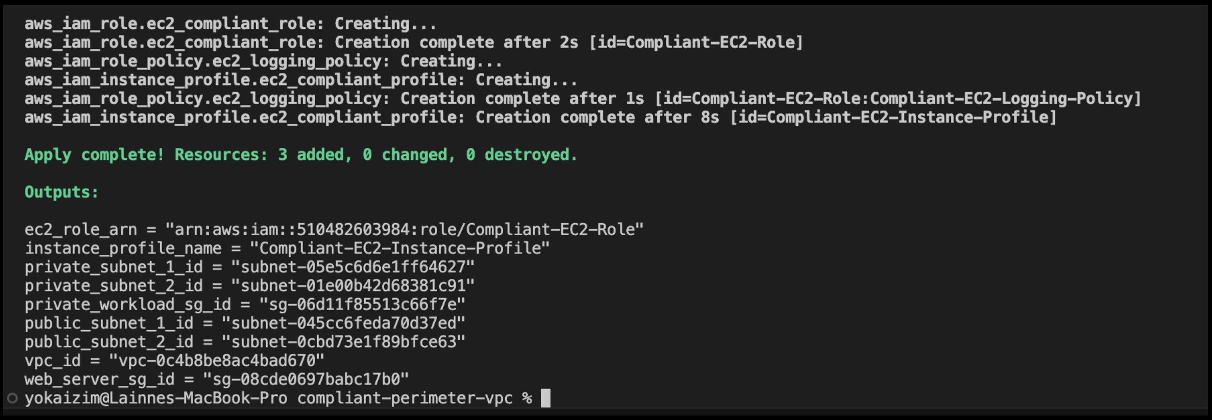
How I'm fixing it: Creating a dedicated IAM Role with an inline policy scoped only to CloudWatch Logs permissions. An Instance Profile is then used to attach this role to the EC2 instance, giving it temporary, auto-rotating credentials at launch.

Why I'm fixing it: This is Least Privilege in practice, not just in theory. The instance gets only the permissions it needs and nothing more. This satisfies NIST CSF PR.AC-4 and ACSC Essential Eight requirements for restricting administrative privileges.



```
aws_iam_role.ec2_compliant_role: Creating...

│ Error: creating IAM Role (Compliant-EC2-Role): operation error IAM: CreateRole, https response error StatusCode: 403,
:510482603984:user/Terraform-Admin is not authorized to perform: iam:CreateRole on resource: arn:aws:iam::510482603984:
│
│   with aws_iam_role.ec2_compliant_role,
│   on iam_role.tf line 40, in resource "aws_iam_role" "ec2_compliant_role":
│   40: resource "aws_iam_role" "ec2_compliant_role" {
│
```

Figure 17: **Terraform apply failed due to missing IAM permissions (403 Unauthorized)**
Terraform-Admin initially lacked the required permissions to create IAM roles and instance profiles, resulting in an AccessDenied error. This highlights the importance of enforcement of AWS IAM boundaries and the need to explicitly authorize privileged actions before infrastructure changes can proceed.



```
aws_iam_role.ec2_compliant_role: Creating...
aws_iam_role.ec2_compliant_role: Creation complete after 2s [id=Compliant-EC2-Role]
aws_iam_role_policy.ec2_logging_policy: Creating...
aws_iam_instance_profile.ec2_compliant_profile: Creating...
aws_iam_role_policy.ec2_logging_policy: Creation complete after 1s [id=Compliant-EC2-Role:Compliant-EC2-Logging-Policy]
aws_iam_instance_profile.ec2_compliant_profile: Creation complete after 8s [id=Compliant-EC2-Instance-Profile]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

ec2_role_arn = "arn:aws:iam::510482603984:role/Compliant-EC2-Role"
instance_profile_name = "Compliant-EC2-Instance-Profile"
private_subnet_1_id = "subnet-05e5c6d6e1ff64627"
private_subnet_2_id = "subnet-01e00b42d68381c91"
private_workload_sg_id = "sg-06d11f85513c66f7e"
public_subnet_1_id = "subnet-045cc6feda70d37ed"
public_subnet_2_id = "subnet-0cbd73e1f89bfce63"
vpc_id = "vpc-0c4b8be8ac4bad670"
web_server_sg_id = "sg-08cde0697babc17b0"
○ yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc %
```

Figure 18:**Successful terraform apply - IAM Role and Instance Profile provisioned**

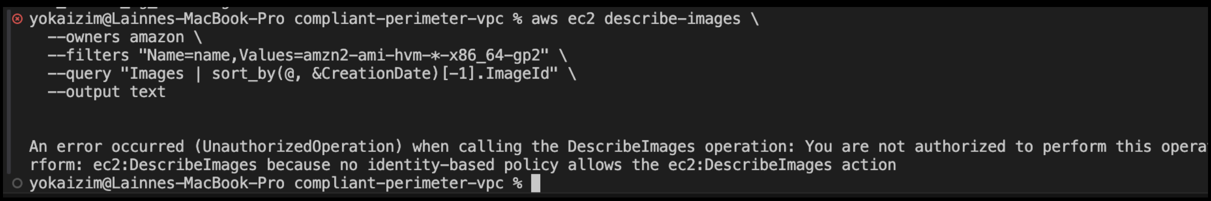
The Compliant-EC2-Role and Instance Profile were created successfully, granting the EC2 instance temporary credentials scoped only to CloudWatch Logs actions.

What I'm fixing: Bridging the gap between IAM permissions and EC2 instances so the server can securely report data without using hardcoded credentials.

How I'm fixing it: Provisioning an IAM Role with a trust relationship for EC2 and an inline policy that grants specific CloudWatch Logs permissions.

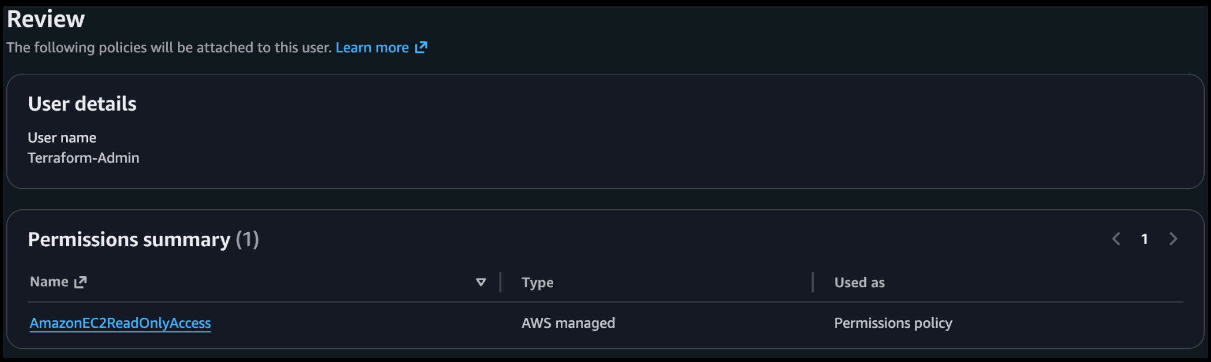
Why I'm fixing it: This adheres to the Principle of Least Privilege and NIST CSF PR.AC-4 by ensuring the instance only has the permissions it needs to perform its logging function and nothing more.

Phase 6: EC2 Deployment



```
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc % aws ec2 describe-images \
  --owners amazon \
  --filters "Name=name,Values=amzn2-ami-hvm-*-x86_64-gp2" \
  --query "Images | sort_by(@, &CreationDate)[-1].ImageId" \
  --output text

An error occurred (UnauthorizedOperation) when calling the DescribeImages operation: You are not authorized to perform this opera
rform: ec2:DescribeImages because no identity-based policy allows the ec2:DescribeImages action
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc %
```

Figure 19: Terraform-Admin required additional EC2 read permissions.



Review

The following policies will be attached to this user. Learn more

User details

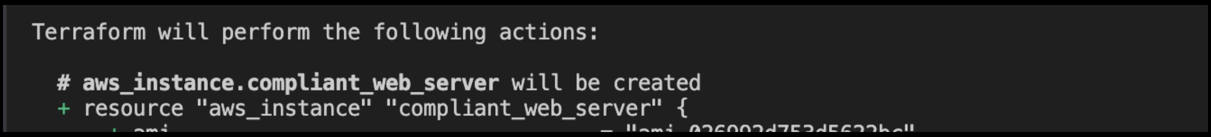
User name
Terraform-Admin

Permissions summary (1)

Name	Type	Used as
AmazonEC2ReadOnlyAccess	AWS managed	Permissions policy

Figure 20: Temporary EC2 Read Permissions Granted

Terraform-Admin required additional EC2 read permissions to retrieve the latest Amazon Linux 2 AMI ID programmatically. This demonstrates AWS enforcement of IAM boundaries and the iterative permission-scoping approach used in DevSecOps workflows.



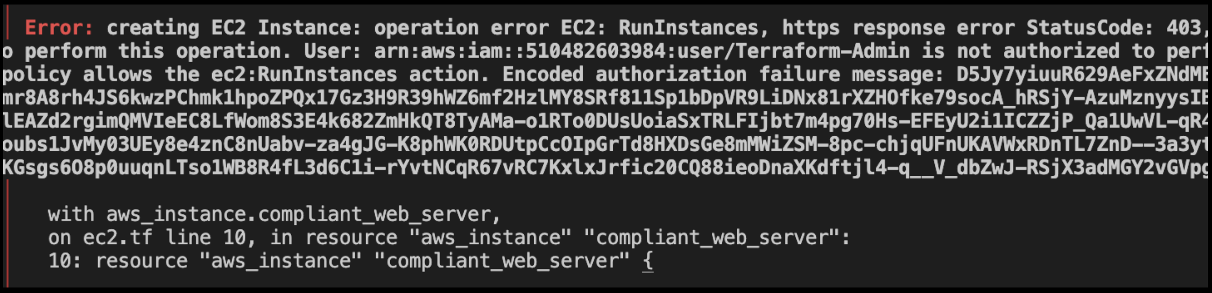
```
Terraform will perform the following actions:

  # aws_instance.compliant_web_server will be created
  + resource "aws_instance" "compliant_web_server" {
```

Figure 21: `terraform plan` output - EC2 Instance

What I'm fixing: The lack of a compute resource to demonstrate the entire compliant perimeter in action. Without an EC2 instance, the security controls (VPC isolation, security groups, IAM role) have nothing to protect.

How I'm fixing it: Provisioning a t3.micro EC2 instance in Public Subnet 1 with the Compliant-WebServer-SG attached and the Compliant-EC2-Instance-Profile for credential management. Public IP auto-assignment is disabled to minimize exposure.

Why I'm fixing it: This completes the demonstration of all three pillars working together - network isolation, micro-segmentation, and least privilege IAM. The instance exists inside the hardened perimeter with only the permissions and network access it needs, satisfying NIST CSF PR.IP-1 and PR.AC-4.



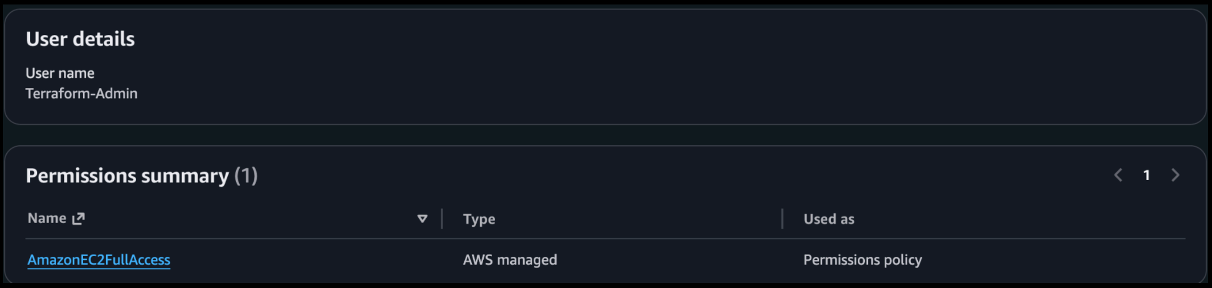
```
Error: creating EC2 Instance: operation error EC2: RunInstances, https response error StatusCode: 403,
o perform this operation. User: arn:aws:iam::510482603984:user/Terraform-Admin is not authorized to perf
policy allows the ec2:RunInstances action. Encoded authorization failure message: D5Jy7yiuuR629AeFxZNdME
mr8A8rh4JS6kwzPChmk1hpoZPQx17Gz3H9R39hWZ6mf2HzlMY8SRf811Sp1bDpVR9LiDNx81rXZHOfke79socA_hRSjY-AzuMznyysIE
lEAZd2rgimQMVIeEC8LfWom8S3E4k682ZmHkQT8TyAMa-o1RTo0DUsUoiaSxTRLFIjbt7m4pg70Hs-EFEyU2i1ICZZjP_Qa1UwVL-qR4
oubs1JvMy03UEy8e4znC8nUabv-za4gJG-K8phWK0RDUtpCcOIpGrTd8HXDsGe8mMWiZSM-8pc-chjqUFnUKAVWxRDnTL7ZnD--3a3yt
KGsgs6O8p0uuqnLTso1WB8R4fL3d6C1i-rYvtNCqR67vRC7KxlxJrfic20CQ88ieoDnaXKdftjl4-q__V_dbZwJ-RSjX3adMGY2vGVpg

    with aws_instance.compliant_web_server,
    on ec2.tf line 10, in resource "aws_instance" "compliant_web_server":
    10: resource "aws_instance" "compliant_web_server" {
```

Figure 22: Terraform apply failed - Missing EC2 launch permissions (403 Unauthorized)

What I'm fixing: Terraform-Admin lacked the `ec2:RunInstances` permission required to actually launch EC2 instances. Read-only EC2 access (granted earlier for AMI lookup) alone is insufficient for provisioning resources.

How I'm fixing it: Attaching the `AmazonEC2FullAccess` managed policy to Terraform-Admin via the AWS Console, granting full EC2 provisioning capabilities.

Why I'm fixing it: This demonstrates AWS's strict enforcement of least privilege at the API level. Each new resource type requires explicit authorization, which will be consolidated into a single scoped policy in Phase 8, showing the iterative hardening approach central to DevSecOps.



User details

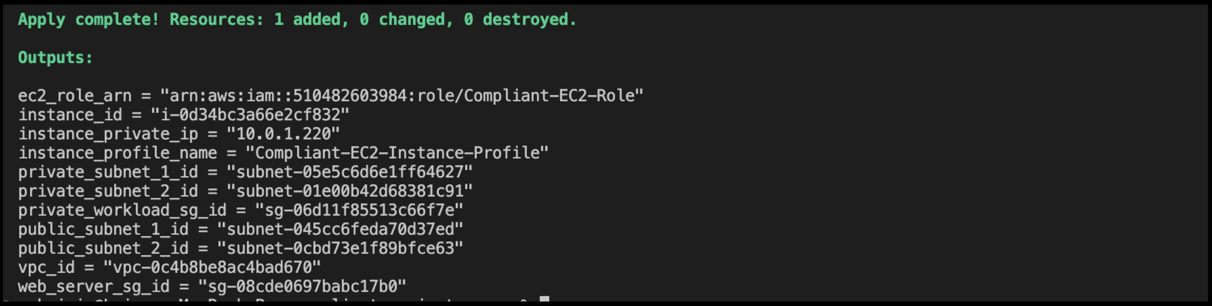
User name
Terraform-Admin

Permissions summary (1)

Name	Type	Used as
AmazonEC2FullAccess	AWS managed	Permissions policy

Figure 23: **AmazonEC2FullAccess policy attached to Terraform-Admin**

Terraform-Admin was granted full EC2 provisioning permissions via the `AmazonEC2FullAccess` managed policy. This unblocks the EC2 instance deployment. As with previous broad policies (VPC, IAM), this will be replaced with a scoped custom policy in Phase 8 to demonstrate the progression from permissive to least-privilege access.



```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

ec2_role_arn = "arn:aws:iam::510482603984:role/Compliant-EC2-Role"
instance_id = "i-0d34bc3a66e2cf832"
instance_private_ip = "10.0.1.220"
instance_profile_name = "Compliant-EC2-Instance-Profile"
private_subnet_1_id = "subnet-05e5c6d6e1ff64627"
private_subnet_2_id = "subnet-01e00b42d68381c91"
private_workload_sg_id = "sg-06d11f85513c66f7e"
public_subnet_1_id = "subnet-045cc6feda70d37ed"
public_subnet_2_id = "subnet-0cbd73e1f89bfce63"
vpc_id = "vpc-0c4b8be8ac4bad670"
web_server_sg_id = "sg-08cde0697babc17b0"
```

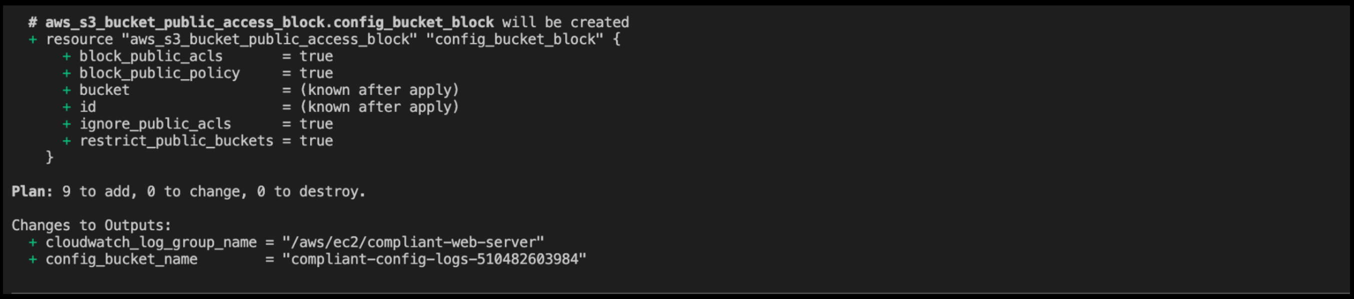
Figure 24: **Successful `terraform apply` - EC2 Instance deployed**

What I'm fixing: Completing the deployment of the compute layer within the compliant perimeter.

How I'm fixing it: The EC2 instance was successfully provisioned in Public Subnet 1 with the Compliant-WebServer-SG and Compliant-EC2-Instance-Profile attached. No public IP was assigned.

Why I'm fixing it: This demonstrates all three security pillars working together - network isolation (VPC + private subnets), micro-segmentation (security groups), and least privilege (IAM role). The instance exists inside a hardened perimeter with only the network access and permissions it needs, satisfying NIST CSF PR.IP-1 and PR.AC-4.

Phase 7: Continuous Monitoring



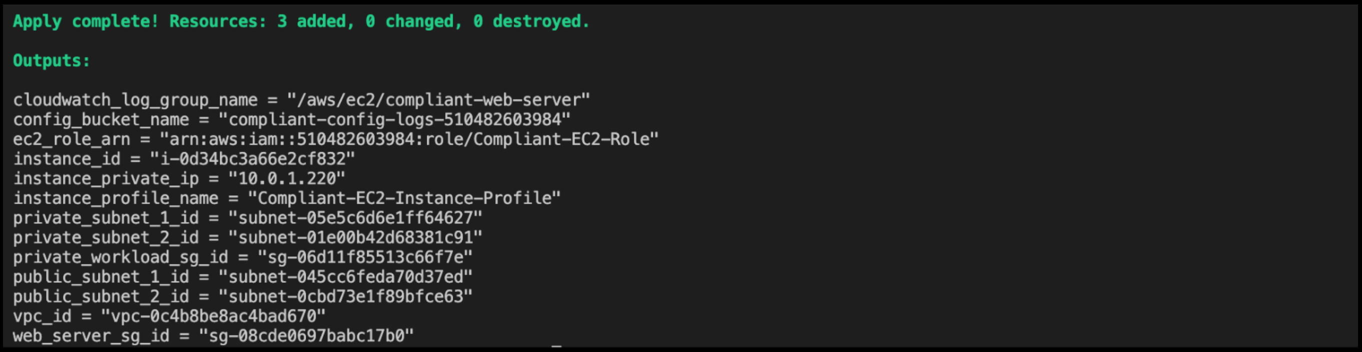
```
  # aws_s3_bucket_public_access_block.config_bucket_block will be created
  + resource "aws_s3_bucket_public_access_block" "config_bucket_block" {
      + block_public_acls       = true
      + block_public_policy     = true
      + bucket                  = (known after apply)
      + id                      = (known after apply)
      + ignore_public_acls      = true
      + restrict_public_buckets = true
    }

Plan: 9 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + cloudwatch_log_group_name = "/aws/ec2/compliant-web-server"
  + config_bucket_name        = "compliant-config-logs-510482603984"
```

Figure 25: `terraform plan` output - Continuous Monitoring (CloudWatch + AWS Config)

What I'm fixing: The lack of visibility into infrastructure changes and system activity. Without monitoring and audit trails, compliance drift and unauthorized changes go undetected.

How I'm fixing it: Provisioning a CloudWatch Log Group for EC2 system logs, an S3 bucket for AWS Config delivery, and an AWS Config Recorder that tracks every resource configuration change in the account.

Why I'm fixing it: This satisfies NIST CSF DE.CM-1 and DE.CM-7 (continuous monitoring) and ISO 27001 A.12.4 (logging and monitoring). AWS Config provides a complete audit trail of all infrastructure changes, ensuring compliance can be verified at any point.

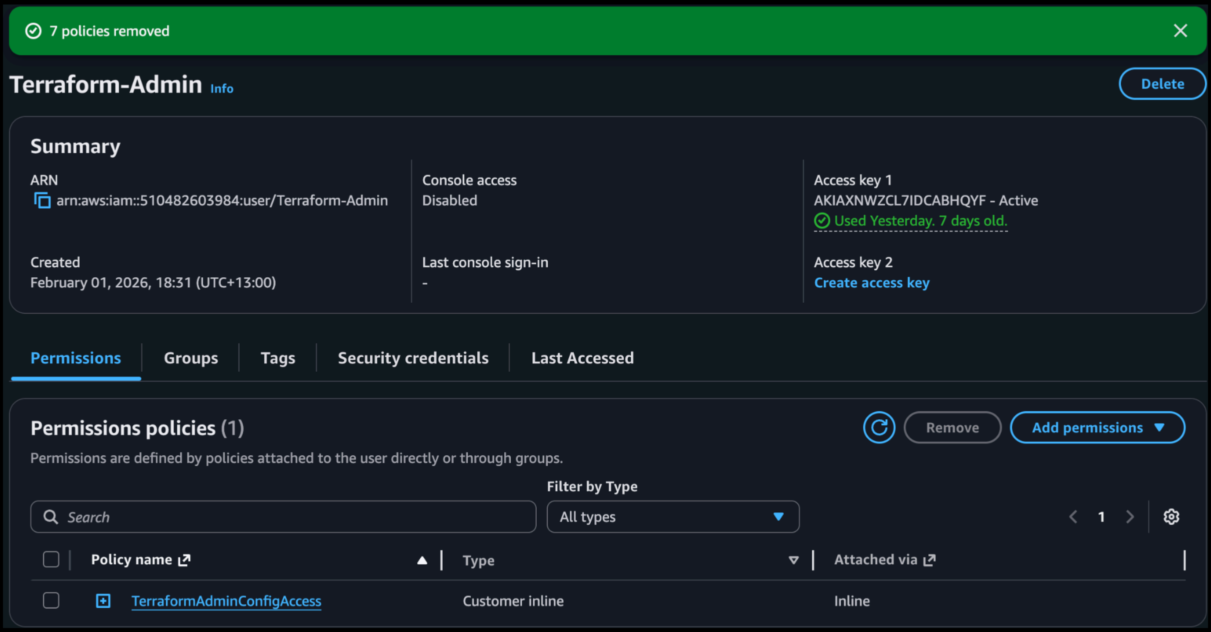


```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

cloudwatch_log_group_name = "/aws/ec2/compliant-web-server"
config_bucket_name = "compliant-config-logs-510482603984"
ec2_role_arn = "arn:aws:iam::510482603984:role/Compliant-EC2-Role"
instance_id = "i-0d34bc3a66e2cf832"
instance_private_ip = "10.0.1.220"
instance_profile_name = "Compliant-EC2-Instance-Profile"
private_subnet_1_id = "subnet-05e5c6d6e1ff64627"
private_subnet_2_id = "subnet-01e00b42d68381c91"
private_workload_sg_id = "sg-06d11f85513c66f7e"
public_subnet_1_id = "subnet-045cc6feda70d37ed"
public_subnet_2_id = "subnet-0cbd73e1f89bfce63"
vpc_id = "vpc-0c4b8be8ac4bad670"
web_server_sg_id = "sg-08cde0697babc17b0"
```

Figure 26: Successful `terraform apply` **- Monitoring infrastructure deployed**

The CloudWatch Log Group, S3 bucket, Config Recorder, and Delivery Channel were created successfully. AWS Config is now actively recording all resource changes and delivering configuration snapshots to the S3 bucket. This completes the fourth pillar of the project - Continuous Monitoring.

Phase 8: Terraform-Admin Policy Hardening



7 policies removed

Terraform-Admin Info

Summary

ARN	Console access	Access key 1
arn:aws:iam::510482603984:user/Terraform-Admin	Disabled	AKIAXNWZCL7IDCABHQYF - Active Used Yesterday. 7 days old.
Created	**Last console sign-in**	**Access key 2**
February 01, 2026, 18:31 (UTC+13:00)	-	Create access key

Permissions | Groups | Tags | Security credentials | Last Accessed

Permissions policies (1)

Permissions are defined by policies attached to the user directly or through groups.

Filter by Type: All types

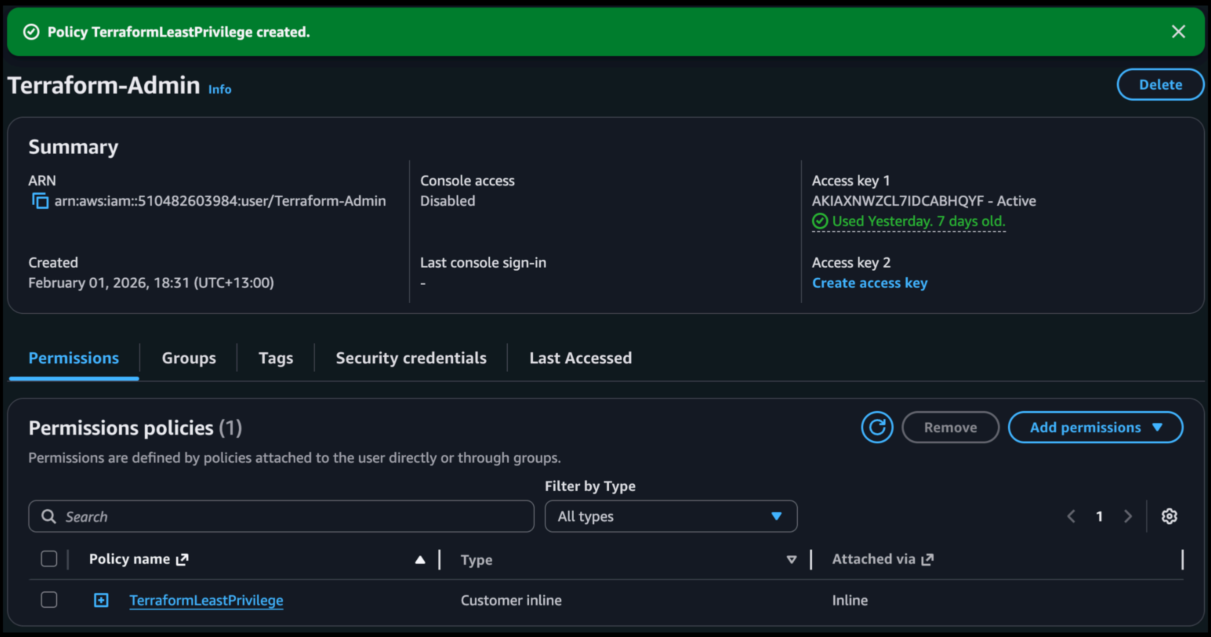
Policy name	Type	Attached via
TerraformAdminConfigAccess	Customer inline	Inline

Figure 27: **All broad managed policies detached from Terraform-Admin**

What I'm fixing: Excessive permissions granted during the iterative build process. The managed policies (VPCFullAccess, EC2FullAccess, IAMFullAccess, S3FullAccess, CloudWatchLogsFullAccess) were necessary to unblock Terraform during deployment, but they grant far more access than required for this specific project.

How I'm fixing it: Detaching all managed policies in preparation for a single custom least-privilege policy that grants only the specific actions Terraform actually used.

Why I'm fixing it: This demonstrates the progression from permissive to least-privilege access, a core tenet of DevSecOps. The final scoped policy will satisfy NIST CSF PR.AC-4 and ACSC Essential Eight by ensuring Terraform-Admin has exactly the permissions it needs and nothing more.


Policy TerraformLeastPrivilege created.

Terraform-Admin

Summary

ARN: arn:aws:iam::510482603984:user/Terraform-Admin

Console access: Disabled

Access key 1: AKIAXNWZCL7IDCABHQYF - Active
Used Yesterday. 7 days old.

Created: February 01, 2026, 18:31 (UTC+13:00)

Last console sign-in: -

Access key 2: Create access key

Permissions policies (1)

Permissions are defined by policies attached to the user directly or through groups.

Filter by Type: All types

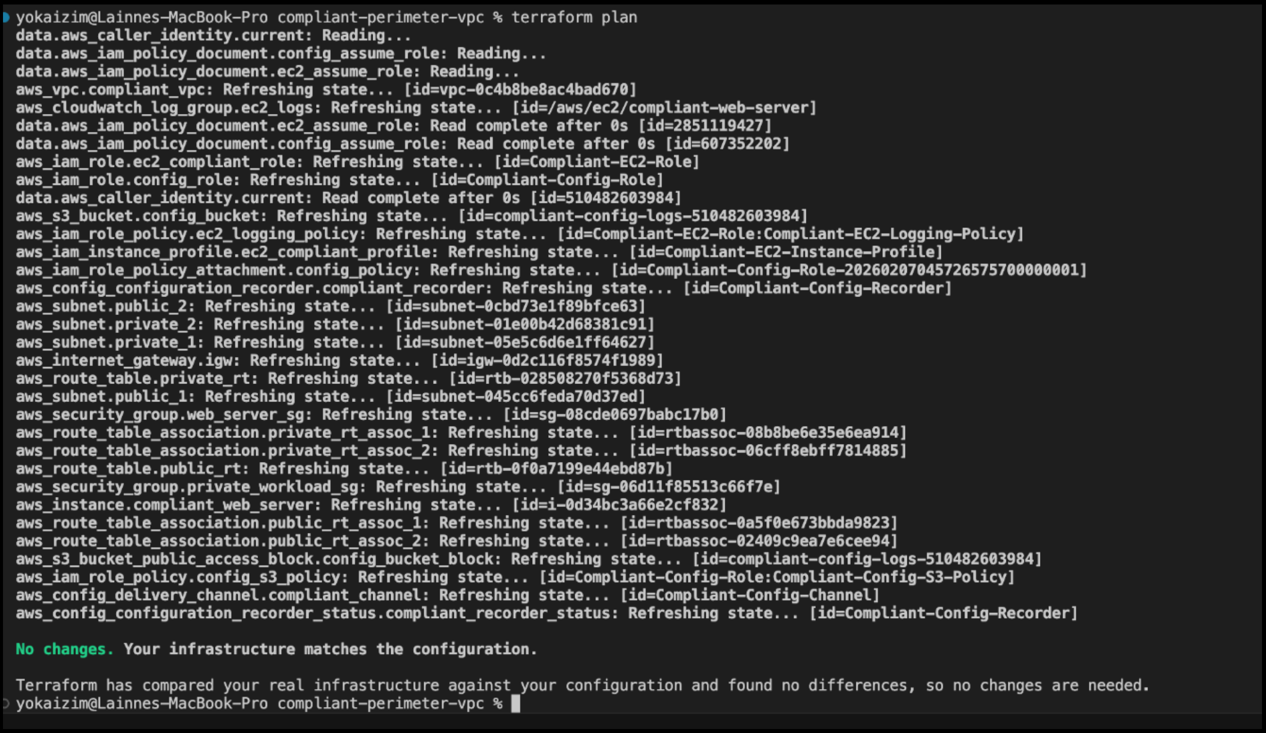
Policy name	Type	Attached via
TerraformLeastPrivilege	Customer inline	Inline

Figure 28: **Scoped least-privilege policy created**

What I'm fixing: The overly permissive managed policies used during the build process. The original policies (VPCFullAccess, EC2FullAccess, etc.) granted far more permissions than Terraform actually needed for this project.

How I'm fixing it: Replacing all broad managed policies with a single custom inline policy that grants only the specific EC2, IAM, S3, CloudWatch Logs, and AWS Config actions used in the deployment. This policy uses targeted wildcards to stay under AWS's 2048-byte inline policy limit while remaining significantly more restrictive than full-access policies.

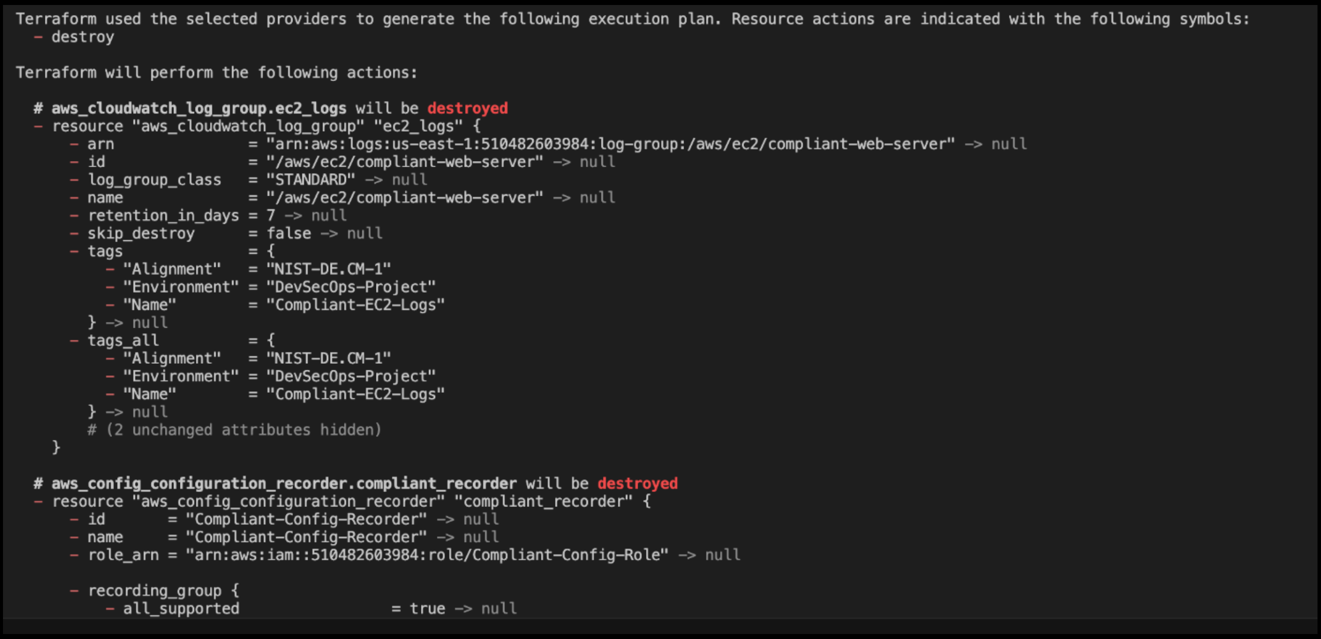
Why I'm fixing it: This demonstrates the DevSecOps principle of iterative hardening - permissive during build, locked down at the end. The final policy satisfies NIST CSF PR.AC-4 and ACSC Essential Eight by ensuring Terraform-Admin has only what it needs to manage this specific infrastructure and nothing more.



```
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc % terraform plan
data.aws_caller_identity.current: Reading...
data.aws_iam_policy_document.config_assume_role: Reading...
data.aws_iam_policy_document.ec2_assume_role: Reading...
aws_vpc.compliant_vpc: Refreshing state... [id=vpc-0c4b8be8ac4bad670]
aws_cloudwatch_log_group.ec2_logs: Refreshing state... [id=/aws/ec2/compliant-web-server]
data.aws_iam_policy_document.ec2_assume_role: Read complete after 0s [id=2851119427]
data.aws_iam_policy_document.config_assume_role: Read complete after 0s [id=607352202]
aws_iam_role.ec2_compliant_role: Refreshing state... [id=Compliant-EC2-Role]
aws_iam_role.config_role: Refreshing state... [id=Compliant-Config-Role]
data.aws_caller_identity.current: Read complete after 0s [id=510482603984]
aws_s3_bucket.config_bucket: Refreshing state... [id=compliant-config-logs-510482603984]
aws_iam_role_policy.ec2_logging_policy: Refreshing state... [id=Compliant-EC2-Role:Compliant-EC2-Logging-Policy]
aws_iam_instance_profile.ec2_compliant_profile: Refreshing state... [id=Compliant-EC2-Instance-Profile]
aws_iam_role_policy_attachment.config_policy: Refreshing state... [id=Compliant-Config-Role-20260207045726575700000001]
aws_config_configuration_recorder.compliant_recorder: Refreshing state... [id=Compliant-Config-Recorder]
aws_subnet.public_2: Refreshing state... [id=subnet-0cbd73e1f89bfce63]
aws_subnet.private_2: Refreshing state... [id=subnet-01e00b42d68381c91]
aws_subnet.private_1: Refreshing state... [id=subnet-05e5c6d6e1ff64627]
aws_internet_gateway.igw: Refreshing state... [id=igw-0d2c116f8574f1989]
aws_route_table.private_rt: Refreshing state... [id=rtb-028508270f5368d73]
aws_subnet.public_1: Refreshing state... [id=subnet-045cc6feda70d37ed]
aws_security_group.web_server_sg: Refreshing state... [id=sg-08cde0697babc17b0]
aws_route_table_association.private_rt_assoc_1: Refreshing state... [id=rtbassoc-08b8be6e35e6ea914]
aws_route_table_association.private_rt_assoc_2: Refreshing state... [id=rtbassoc-06cff8ebff7814885]
aws_route_table.public_rt: Refreshing state... [id=rtb-0f0a7199e44ebd87b]
aws_security_group.private_workload_sg: Refreshing state... [id=sg-06d11f85513c66f7e]
aws_instance.compliant_web_server: Refreshing state... [id=i-0d34bc3a66e2cf832]
aws_route_table_association.public_rt_assoc_1: Refreshing state... [id=rtbassoc-0a5f0e673bbda9823]
aws_route_table_association.public_rt_assoc_2: Refreshing state... [id=rtbassoc-02409c9ea7e6cee94]
aws_s3_bucket_public_access_block.config_bucket_block: Refreshing state... [id=compliant-config-logs-510482603984]
aws_iam_role_policy.config_s3_policy: Refreshing state... [id=Compliant-Config-Role:Compliant-Config-S3-Policy]
aws_config_delivery_channel.compliant_channel: Refreshing state... [id=Compliant-Config-Channel]
aws_config_configuration_recorder_status.compliant_recorder_status: Refreshing state... [id=Compliant-Config-Recorder]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
yokaizim@Lainnes-MacBook-Pro compliant-perimeter-vpc %
```

Figure 29: `terraform plan` **succeeds with scoped policy**

Terraform successfully validated the infrastructure using only the scoped `TerraformLeastPrivilege` policy. This confirms the custom policy grants sufficient permissions for all project resources (VPC, EC2, IAM, S3, CloudWatch, Config) without the broad access of managed policies like FullAccess. The policy is scoped to only 5 AWS service namespaces instead of hundreds.



```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # aws_cloudwatch_log_group.ec2_logs will be destroyed
  - resource "aws_cloudwatch_log_group" "ec2_logs" {
      - arn               = "arn:aws:logs:us-east-1:510482603984:log-group:/aws/ec2/compliant-web-server" -> null
      - id                = "/aws/ec2/compliant-web-server" -> null
      - log_group_class   = "STANDARD" -> null
      - name              = "/aws/ec2/compliant-web-server" -> null
      - retention_in_days = 7 -> null
      - skip_destroy      = false -> null
      - tags              = {
          - "Alignment"   = "NIST-DE.CM-1"
          - "Environment" = "DevSecOps-Project"
          - "Name"        = "Compliant-EC2-Logs"
        } -> null
      - tags_all          = {
          - "Alignment"   = "NIST-DE.CM-1"
          - "Environment" = "DevSecOps-Project"
          - "Name"        = "Compliant-EC2-Logs"
        } -> null
        # (2 unchanged attributes hidden)
    }

  # aws_config_configuration_recorder.compliant_recorder will be destroyed
  - resource "aws_config_configuration_recorder" "compliant_recorder" {
      - id       = "Compliant-Config-Recorder" -> null
      - name     = "Compliant-Config-Recorder" -> null
      - role_arn = "arn:aws:iam::510482603984:role/Compliant-Config-Role" -> null

      - recording_group {
          - all_supported                 = true -> null
```

Figure 30: `terraform destroy` **completed - All infrastructure decommissioned**

What I'm fixing: Residual cloud resources that could pose a security or cost risk if left running after project completion.

How I'm fixing it: Running `terraform destroy` to programmatically tear down all 26 provisioned resources (VPC, subnets, route tables, security groups, EC2 instance, IAM roles, S3 bucket, CloudWatch logs, AWS Config recorder). The S3 bucket required manual emptying before deletion due to AWS protection against accidental data loss.

Why I'm fixing it: This adheres to the NIST 800-88 standard for media sanitization and asset disposal mentioned in the executive summary. By destroying all infrastructure through code, we ensure zero residual risk - there are no keys to a kingdom that no longer exists. This demonstrates the full lifecycle of IaC: build, harden, audit, and decommission.