IaC Security

Table of Contents

How do you secure sensitive information in IaC? . . . 1
How do you secure secrets and sensitive variables in Terraform? . . . 2
How would you implement least privilege when defining IAM roles and policies in Terraform? . . . 2
How do you implement least privilege in a cloud environment? . . . 2
What are some best practices for state file management in Terraform? . . . 3
How can policy-as-code tools like Open Policy Agent (OPA) or HashiCorp Sentinel help in IaC security? . . . 3
Describe how you'd enforce security policies as code in an IaC workflow. . . . 3
How do you ensure compliance with IaC? . . . 4
What are common misconfigurations that lead to cloud breaches? . . . 4
Explain the shared responsibility model in the context of cloud security. . . . 5
What is the purpose of `terraform plan` in Terraform? . . . 5
What's the difference between terraform plan and terraform apply in a secure CI/CD pipeline? . . . 5
How do you review and approve Terraform changes in a secure way? . . . 6
How do you embed security checks in a CI/CD pipeline that deploys Terraform code? . . . 6
How do you integrate Terraform with security tools like Checkov, tfsec, or Sentinel? . . . 7
How would you prevent accidental data exposure when using Terraform with cloud storage (like S3 buckets)? . . . 7
How would you secure access to cloud management consoles? . . . 7
What steps would you take to secure public-facing cloud resources? . . . 8
A junior developer committed a plaintext AWS access key to GitHub — how would you detect and respond? . . . 8
Your Terraform code creates a VPC with open security groups — how would you catch that before deployment? . . . 9
You're onboarding a new cloud account — how would you use Terraform to establish baseline security? . . . 9
Show a Terraform snippet to create an S3 bucket with proper encryption and block public access. . . . 10
Walk through how you'd use a custom module to deploy secure EC2 instances with Terraform. . . . 11

How do you secure sensitive information in IaC?

I approach this through multiple layers. First, I never commit secrets directly to version control—instead, I use secret management systems like AWS Secrets Manager, HashiCorp Vault, or cloud-

native KMS services. In the code itself, I reference these secrets by ID rather than value.

I also implement encryption at rest for any state files, use environment variables or CI/CD secret stores for credentials, and apply RBAC to limit who can access the IaC repositories. Additionally, I use tools like git-secrets or Gitleaks in pre-commit hooks to catch accidental secret commits before they reach the repository.

How do you secure secrets and sensitive variables in Terraform?

I use several methods depending on the environment. For sensitive values, I mark them with `sensitive = true` in variable definitions to prevent them from appearing in logs or console output. I store actual secret values in external secret managers like AWS Secrets Manager or Vault, then reference them using data sources.

For CI/CD pipelines, I inject secrets as environment variables prefixed with `TF_VAR_`. I also encrypt the Terraform state file since it stores resource details in plaintext—using S3 with server-side encryption and DynamoDB for state locking, or Terraform Cloud's encrypted state storage. Never hardcode secrets or use default values for sensitive variables.

How would you implement least privilege when defining IAM roles and policies in Terraform?

I start by defining the minimum permissions needed for each role to function, avoiding wildcard actions and resources wherever possible. I use condition statements to further restrict when and how permissions can be used—like limiting access to specific IP ranges or requiring MFA. I create custom policies rather than attaching AWS managed policies that are often too broad.

I also regularly use IAM Access Analyzer to identify unused permissions and refine policies. In Terraform, I organize roles by service or function, document why each permission is needed, and implement periodic reviews through automated tools that flag overly permissive configurations before they're deployed.

How do you implement least privilege in a cloud environment?

Beyond IAM policies, I implement least privilege across multiple dimensions:

- I use **separate accounts or projects** for different environments and workloads, applying service control policies or organizational policies at the top level.
- **Network segmentation** with security groups and NACLs limits lateral movement.

- I enable **resource-based policies** to control access from specific sources.
- For compute resources, I use **instance profiles or workload identity** rather than long-lived credentials.
- I implement **just-in-time access** for administrative tasks, require MFA for privileged operations, and maintain detailed audit logs.

Regular access reviews and automated policy validation ensure drift doesn't occur over time.

What are some best practices for state file management in Terraform?

State files are critical and contain sensitive data, so I treat them like production secrets. I always use **remote state with encryption**--S3 with KMS encryption and versioning enabled, plus DynamoDB for state locking to prevent concurrent modifications.

I restrict access to the state backend using IAM policies that follow least privilege. I enable state file versioning for rollback capability and **never commit state files to version control**. For team environments, I implement proper RBAC on the remote backend and consider using Terraform Cloud or Enterprise for enhanced state management with built-in encryption, versioning, and access controls. Regular state backups to a separate location provide disaster recovery capability.

How can policy-as-code tools like Open Policy Agent (OPA) or HashiCorp Sentinel help in IaC security?

These tools act as guardrails that enforce security standards automatically. With OPA or Sentinel, I write policies that check for common misconfigurations before infrastructure is deployed—things like ensuring S3 buckets aren't public, requiring encryption at rest, or verifying security groups don't allow unrestricted ingress.

These policies run during `terraform plan` or in CI/CD pipelines, failing the deployment if violations are found. This **shifts security left** by catching issues at code review rather than in production. I can also create policies that enforce organizational standards like required tags, approved instance types, or mandatory backup configurations. The policies themselves are versioned and tested, creating a compliance-as-code approach that's repeatable and auditable.

Describe how you'd enforce security policies as code in an IaC workflow.

I integrate policy enforcement at multiple stages:

- **Development phase**: IDE plugins that lint Terraform code against security policies in real-time.

- **Pre-commit hooks**: run tools like tfsec or Checkov locally before code reaches version control.
- **CI/CD pipeline**: dedicated security scanning stages that run after `terraform plan` but before human review—these use multiple tools for broader coverage.
- **Policy enforcement**: OPA or Sentinel policies for custom organizational rules. Failed policy checks block the pipeline and provide detailed reports on violations.
- **Exceptions**: documented override process that requires security team approval and is tracked in an audit log.

All policies are versioned alongside infrastructure code and reviewed regularly.

How do you ensure compliance with IaC?

Compliance starts with encoding requirements directly into Terraform modules and policies. I map compliance frameworks like SOC 2, HIPAA, or PCI-DSS to specific infrastructure controls, then implement those as reusable modules and policy checks. I use automated scanning tools that check against CIS benchmarks and other standards.

All infrastructure changes go through peer review with security-focused checklists. I maintain detailed documentation linking infrastructure code to specific compliance requirements. Terraform outputs and tags help with compliance reporting and resource tracking. I implement drift detection to catch out-of-band changes that could violate compliance. Regular compliance audits review both the code and deployed infrastructure, with findings fed back into policy improvements.

What are common misconfigurations that lead to cloud breaches?

The most frequent issues I see are:

- **Publicly accessible storage buckets** — S3 buckets with open ACLs or bucket policies allowing anonymous access.
- **Overly permissive security groups** allowing SSH or RDP from 0.0.0.0/0.
- **Disabled or insufficient logging** makes it hard to detect breaches.
- **Lack of encryption** for data at rest and in transit.
- **Overly broad IAM policies** with wildcard permissions or attached to users instead of roles.
- **Disabled MFA** on privileged accounts.
- **Exposed secrets** in code or logs.
- **Unpatched instances** with known vulnerabilities.
- **Lack of network segmentation** allowing lateral movement.

All these create attack vectors that threat actors actively exploit.

Explain the shared responsibility model in the context of cloud security.

The cloud provider and customer split security responsibilities:

- **The provider secures the infrastructure** — physical data centers, hypervisors, network hardware, and managed service components.
- **As the customer, I'm responsible for security *in* the cloud** — my data, applications, operating systems, network configurations, IAM policies, and encryption.

For managed services, the division shifts:

- With **EC2**, I manage everything from the OS up.
- With **RDS**, AWS handles OS patching but I manage database credentials and access controls.
- With **S3**, AWS secures the storage infrastructure but I configure bucket policies and encryption.

Understanding this boundary is critical—I can't assume the cloud provider secures things like security groups or IAM policies, those are squarely my responsibility.

What is the purpose of `terraform plan` in Terraform?

`terraform plan` creates an execution plan showing what changes Terraform will make to reach the desired state defined in configuration files. It compares the current state with the desired state and shows additions, modifications, and deletions without actually applying them.

From a security perspective, this is my primary review checkpoint. I examine the plan output for unexpected changes, resources being destroyed, or configuration changes that might introduce security issues. In automated workflows, the plan output is what security tools analyze and what human reviewers approve before deployment. It's essentially a preview that lets me catch errors or security issues before they impact production infrastructure.

What's the difference between terraform plan and terraform apply in a secure CI/CD pipeline?

In a secure pipeline, these represent different stages with different security controls:

- `terraform plan` **runs first** and generates a plan file that's saved as an artifact. This plan goes through security scanning—tools like tfsec, Checkov, or Sentinel analyze it for policy violations. Human reviewers examine the plan for unexpected changes or security concerns. Only after all security gates pass does the plan get approved for application.

- `terraform apply` **then executes** the specific approved plan file using the `-auto-approve` flag with the saved plan.

This separation ensures what was reviewed is exactly what gets applied. I also implement additional controls like requiring multiple approvers for production changes or time-gating applications to specific deployment windows.

How do you review and approve Terraform changes in a secure way?

I implement a multi-stage approval process:

- **Code changes** start with peer review in pull requests, where developers check for functionality and obvious security issues using checklists.
- **Automated security scanning** runs on every PR, blocking merge if critical issues are found.
- **Plan review** in a staging or pre-production environment where security and operations teams review the actual changes that will occur.
- **Production approvals** require explicit approval from designated approvers—often requiring multiple approvals for high-risk changes.
- **Plan security**: The approved plan file is cryptographically signed or stored in a secure artifact repository.
- **Audit trail**: All approvals are logged with timestamps and approver identities.

For particularly sensitive changes, I schedule them during change windows with additional monitoring and rollback procedures in place.

How do you embed security checks in a CI/CD pipeline that deploys Terraform code?

I create dedicated security stages in the pipeline:

- **After code checkout**: static analysis with multiple tools—tfsec for Terraform-specific checks, Checkov for policy validation, and custom scripts for organization-specific rules. I use Trivy or similar tools to scan for vulnerabilities in any container images or dependencies.
- **After** `terraform plan`: parse the output and run additional checks on the proposed changes. Integrate with secret scanning tools to ensure no credentials are in the code.
- **Before apply**: manual approval gate for production deployments.
- **Post-deployment**: trigger compliance scans against the actual deployed resources and send results to security dashboards.

Failed security checks fail the pipeline with detailed reports. I also implement drift detection jobs that run periodically to catch out-of-band changes.

How do you integrate Terraform with security tools like Checkov, tfsec, or Sentinel?

- For **tfsec and Checkov**, I integrate them as pipeline stages that run against the Terraform code directory. They scan for misconfigurations and output results in various formats—I typically use JSON for parsing in automation and JUnit for CI/CD integration. Critical severity findings fail the pipeline.
- For **Sentinel with Terraform Cloud or Enterprise**, I write policies in the Sentinel language and attach them to workspaces, configuring which policies are advisory versus mandatory.
- **Locally**, developers can run these tools in pre-commit hooks for immediate feedback.

I maintain a central repository of security policies that's versioned and tested, with documentation explaining each rule and any approved exceptions. Results feed into security dashboards for tracking trends and identifying systemic issues across teams.

How would you prevent accidental data exposure when using Terraform with cloud storage (like S3 buckets)?

I create Terraform modules with secure defaults—block public access at both the bucket and account level, require encryption with KMS, enable versioning, and enforce bucket policies that deny unencrypted uploads. In my modules, I explicitly set `block_public_acls`, `block_public_policy`, `ignore_public_acls`, and `restrict_public_buckets` all to true.

I use bucket policies that require encryption in transit and restrict access to specific IAM roles or VPCs. I implement automated scanning using tools like Prowler or Scout Suite that detect publicly accessible buckets immediately after creation. In CI/CD, Checkov or tfsec rules fail deployments that would create public buckets. I also enable AWS Access Analyzer to continuously monitor for external access. Any bucket requiring public access goes through an exception process with security review and additional compensating controls.

How would you secure access to cloud management consoles?

I implement multiple layers of access control:

- **Enforce MFA** for all console access—no exceptions.
- **Single sign-on** with SAML integration to centralize authentication and enable conditional access policies.

- **Role-based access**: Access is granted through role assumption rather than long-lived credentials, with session durations limited to necessary time periods.
- **IP allowlisting** where feasible, restricting console access to corporate networks or VPN endpoints.
- **Step-up authentication** for highly privileged operations.
- **Comprehensive logging**: All console activities are logged to CloudTrail or equivalent and monitored for suspicious patterns.
- **Root account security**: Disable root account access keys and use the root account only for break-glass scenarios with alerts on any usage.

Regular access reviews ensure users only have necessary permissions, and I implement automatic session timeouts and account lockouts after failed login attempts.

What steps would you take to secure public-facing cloud resources?

I start with the principle that resources should be private by default, making things public only when absolutely necessary. For truly public resources like websites, I place them behind CDN services like CloudFront that provide DDoS protection and WAF integration. I implement strict security groups allowing only required ports and protocols.

For web applications, I use WAF rules to filter malicious traffic and protect against OWASP Top 10 vulnerabilities. I enable logging at every layer—load balancer logs, application logs, WAF logs. I implement TLS 1.2 or higher with strong cipher suites.

Regular vulnerability scanning and penetration testing identify issues before attackers do. I use rate limiting and throttling to prevent abuse. Network architecture includes multiple availability zones with auto-scaling for resilience. I also implement monitoring and alerting for anomalous traffic patterns and automated response playbooks for common attack scenarios.

A junior developer committed a plaintext AWS access key to GitHub — how would you detect and respond?

For **detection**, I rely on multiple layers:

- GitHub secret scanning should catch it immediately and notify us.
- git-secrets or Gitleaks in pre-commit hooks (though they apparently didn't run here).
- AWS GuardDuty would detect unusual API activity from the exposed key.

For **response**:

- **Immediately invalidate** the exposed credentials through IAM—delete or rotate the access key

within minutes of detection.

- **Check CloudTrail logs** for any API calls made with those credentials to understand the blast radius.
- If unauthorized activity occurred, **treat it as a security incident**--contain affected resources, conduct forensics, and determine what data or systems were accessed.
- **Blameless postmortem** to understand why preventive controls failed and implement improvements—enforcing pre-commit hooks, adding CI/CD secret scanning, improving developer training on secret management, and potentially implementing AWS credentials vending systems that eliminate long-lived keys.

Your Terraform code creates a VPC with open security groups — how would you catch that before deployment?

I catch this through multiple checkpoints:

- **During development**: The developer should run tfsec or Checkov locally, which flag security groups allowing 0.0.0.0/0 on sensitive ports.
- **In the pull request**: Automated CI checks run these same tools and comment findings directly on the PR, failing the build if critical issues exist.
- **Security review**: The security team reviews the PR with a focus on networking and access controls.
- **Plan review**: When `terraform plan` runs, the output shows the security group rules being created—human reviewers specifically look for overly permissive ingress rules.
- **Policy-as-code**: Tools like Sentinel can enforce rules preventing security groups with 0.0.0.0/0 on non-standard ports.
- **Secure modules**: I maintain Terraform modules for common patterns with secure defaults, so developers using those modules wouldn't create this issue in the first place.
- **Post-deployment**: Automated compliance scans would catch it as drift if it somehow made it through, triggering alerts and automated remediation.

You're onboarding a new cloud account — how would you use Terraform to establish baseline security?

I use Terraform to implement a security baseline as the first step in account setup. I'd start with a foundational module that includes:

- **Logging & Monitoring**: Enable CloudTrail with log file validation, AWS Config, GuardDuty, and Security Hub.

- **IAM Security**: Set up password policy with MFA requirements, initial IAM structure with SSO integration.
- **Network Visibility**: Establish VPC flow logs.
- **Data Protection**: Enable S3 block public access at the account level, configure KMS with key rotation.
- **Governance**: Set up billing alarms and tag policies.

All of this would be in versioned Terraform code that serves as the template for all new accounts, ensuring consistent security posture across the organization.

Show a Terraform snippet to create an S3 bucket with proper encryption and block public access.

```
resource "aws_s3_bucket" "secure_bucket" {
  bucket = "example-secure-bucket"

  tags = {
    Environment = "production"
    ManagedBy   = "terraform"
  }
}

resource "aws_s3_bucket_versioning" "secure_bucket" {
  bucket = aws_s3_bucket.secure_bucket.id

  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "secure_bucket" {
  bucket = aws_s3_bucket.secure_bucket.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm     = "aws:kms"
      kms_master_key_id = aws_kms_key.bucket_key.arn
    }
    bucket_key_enabled = true
  }
}

resource "aws_s3_bucket_public_access_block" "secure_bucket" {
  bucket = aws_s3_bucket.secure_bucket.id
```

```
  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}

resource "aws_s3_bucket_logging" "secure_bucket" {
  bucket = aws_s3_bucket.secure_bucket.id

  target_bucket = aws_s3_bucket.log_bucket.id
  target_prefix = "access-logs/"
}

resource "aws_kms_key" "bucket_key" {
  description             = "KMS key for S3 bucket encryption"
  deletion_window_in_days = 10
  enable_key_rotation     = true
}
```

This demonstrates:

- Encryption with KMS
- Versioning for recovery
- Complete public access blocking
- Access logging for audit trails
- Key rotation for security best practices

Walk through how you'd use a custom module to deploy secure EC2 instances with Terraform.

I'd create a module at `modules/secure-ec2` that encapsulates security best practices. The module would require minimal inputs—instance type, AMI ID, subnet ID—while enforcing secure defaults.

Inside the module, I'd:

- Create the instance with an **IAM instance profile** (no hardcoded credentials)
- Associate it with a **security group** with least-privilege rules
- Enable **detailed monitoring**
- Encrypt the **root volume with KMS**
- Require instances to be launched in **private subnets**
- Use **Systems Manager for access** instead of SSH keys
- Include **user data** that installs security agents, configures logging, and applies OS-level

hardening

The module would output the instance ID and private IP but not expose anything sensitive. To use it, teams would call the module with their specific variables, knowing security controls are built-in.

I'd version the module, maintain documentation with security justifications for each configuration, and require security team review for module changes. This way, secure EC2 deployment becomes the path of least resistance for developers.