Securing Infrastructure as Code

A comprehensive guide to secrets management, policy enforcement, and security scanning for cloud infrastructure.

The Problem with Secrets

Infrastructure code requires access to sensitive data like database passwords, API keys, and certificates. Hardcoding these values directly into code creates serious security risks.

Imagine checking your AWS access key into a public GitHub repository. Within minutes, automated bots could discover it and rack up thousands in unauthorized charges.

Three Approaches to Secrets Management

Each approach offers different trade-offs between security, complexity, and ease of use. Let's explore how each works in practice.

Environment Variables

Simple and quick for development environments

- Easy to implement locally
- Limited security features
- Best for non-production

AWS Secrets Manager

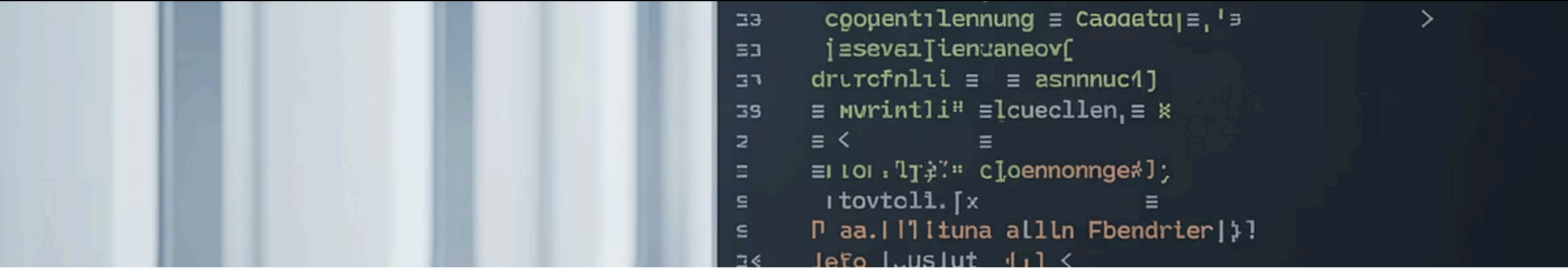
Native AWS service with automatic rotation

- Integrated with AWS services
- Automatic secret rotation
- Pay per secret stored

HashiCorp Vault

Enterprise-grade secrets management platform

- Multi-cloud support
- Dynamic secrets generation
- Advanced access controls



Environment Variables: The Starting Point

Environment variables are the simplest secrets management approach. They work well for local development and testing but have limitations in production environments.

You set variables outside your code, then reference them within your Terraform configurations. For example, you might store `DB_PASSWORD` as an environment variable and access it through Terraform's variable system.

The main advantage is simplicity—no additional tools required. However, environment variables can be exposed through process listings, logs, or error messages. They also don't provide audit trails or rotation capabilities.

Environment Variables in Practice

Setting the Variable

```
export TF_VAR_db_password="SuperSecret123"
export TF_VAR_api_key="abc-xyz-789"
```

Variables prefixed with `TF_VAR_` are automatically available to Terraform.

Using in Terraform

```
variable "db_password" {
  type      = string
  sensitive = true
}

resource "aws_db_instance" "main" {
  password = var.db_password
}
```

Best Practice: Never commit environment variable files like `.env` to version control. Add them to your `.gitignore` file immediately.

AWS Secrets Manager: Cloud-Native Solution

AWS Secrets Manager provides centralized secrets storage with built-in security features. It integrates seamlessly with other AWS services and handles common challenges like credential rotation automatically.

Think of it as a secure vault specifically designed for AWS environments. You store a secret once, and any authorized service or user can retrieve it. The service maintains detailed audit logs of every access attempt.

Secrets Manager can automatically rotate credentials on a schedule you define. For example, your RDS database password could rotate every 30 days without manual intervention or application downtime.

Storing and Retrieving Secrets in AWS

Create the Secret

Store your sensitive data in AWS Secrets Manager through the console, CLI, or Terraform itself.

```
aws secretsmanager create-secret \
  --name prod/db/password \
  --secret-string "MySecurePassword"
```

Reference in Terraform

Use the AWS data source to retrieve the secret value securely during deployment.

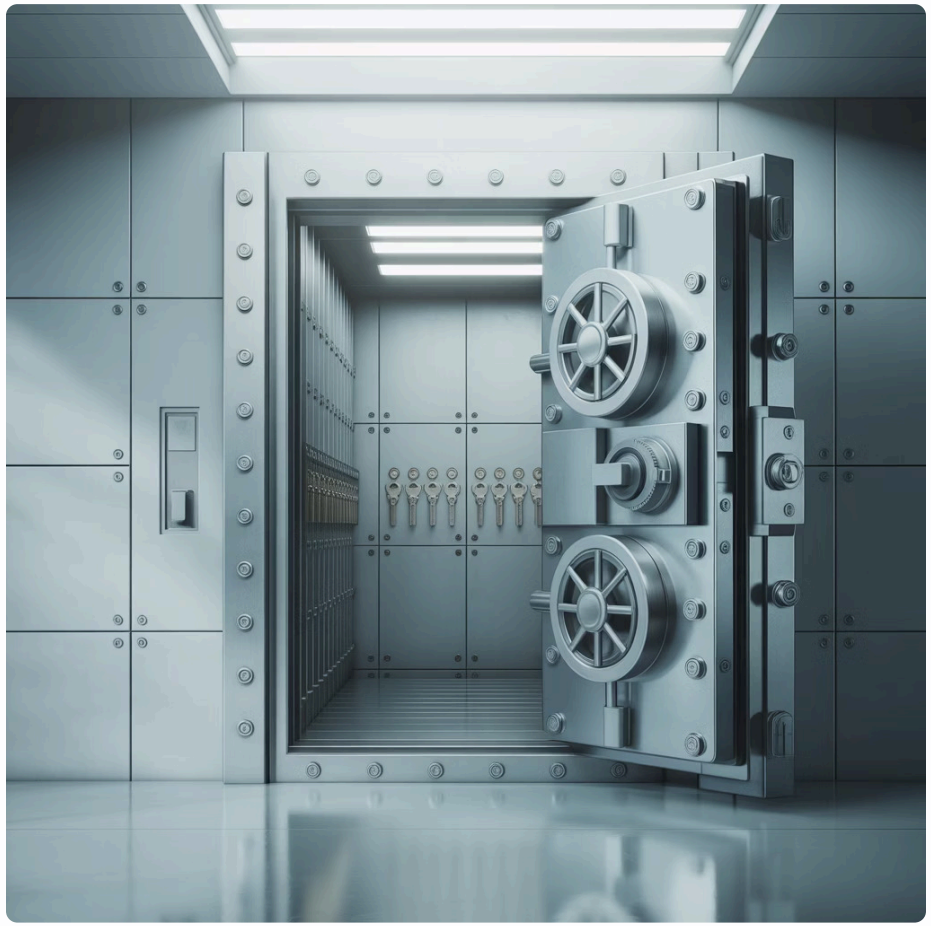
```
data "aws_secretsmanager_secret_version" "db" {
 secret_id = "prod/db/password"
}

resource "aws_db_instance" "main" {
 password =
data.aws_secretsmanager_secret_version.db.secret_string
}
```

HashiCorp Vault: Enterprise Power

HashiCorp Vault goes beyond simple secret storage to provide a comprehensive secrets management platform. It works across multiple cloud providers and on-premises infrastructure.

Vault's standout feature is dynamic secrets—it can generate credentials on-demand and automatically revoke them after use. For instance, instead of sharing a permanent database password, Vault creates temporary credentials that expire after one hour.

This approach dramatically reduces risk. Even if credentials are compromised, they're already invalid by the time an attacker discovers them.



Vault Dynamic Secrets in Action

Application Requests Access

1

Your Terraform code authenticates to Vault and requests database credentials for deployment.

Vault Generates Credentials

2

Vault creates a new database user with specific permissions and a 1-hour expiration time.

Terraform Uses Credentials

3

Your infrastructure code uses the temporary credentials to configure resources.

Automatic Revocation

4

After one hour, Vault automatically deletes the temporary database user, eliminating the security risk.

Terraform Sensitive Variables

Terraform provides built-in protection for sensitive values through the `sensitive` parameter. This prevents secrets from appearing in terminal output, logs, or state files in plain text.

When you mark a variable as sensitive, Terraform replaces its value with `(sensitive value)` in all output. This protects against accidental exposure during `terraform plan` or `terraform apply` commands.

However, sensitive variables alone aren't a complete security solution. They prevent casual exposure but don't encrypt state files or provide access controls. Combine them with proper secrets management tools for comprehensive protection.

Implementing Sensitive Variables

Variable Declaration

```
variable "database_password" {
  description = "Master password for RDS"
  type        = string
  sensitive   = true
}

variable "api_token" {
  description = "Third-party API token"
  type        = string
  sensitive   = true
}
```

What You See in Output

```
Terraform will perform the following actions:

  # aws_db_instance.main will be created
  + resource "aws_db_instance" "main" {
      + password = (sensitive value)
      + username = "admin"
    }
```

Notice how the password value is masked while non-sensitive values display normally.

Policy as Code

Automating governance and compliance for infrastructure deployments

Why Policy as Code Matters

Manual review processes can't keep pace with modern infrastructure deployment speeds. Teams might deploy hundreds of resources daily across multiple cloud accounts.

The Traditional Challenge

A developer submits infrastructure code for review. A senior engineer manually checks naming conventions, required tags, and security settings. This takes hours or days, creating bottlenecks.

The Policy as Code Solution

Automated policies check every deployment instantly. Rules enforce naming standards, required tags, cost controls, and security configurations without human intervention.

Policy as code transforms compliance from a manual checkpoint into an automated safety net. Deployments that violate policies fail immediately with clear error messages explaining what needs to change.

Two Policy Frameworks

Sentinel (Terraform Cloud)

Sentinel is HashiCorp's policy-as-code framework, deeply integrated with Terraform Cloud and Enterprise.

- Native Terraform integration
- Access to plan and state data
- Three enforcement levels
- Commercial product

OPA (Open Policy Agent)

OPA is an open-source, general-purpose policy engine from the Cloud Native Computing Foundation.

- Works with any data source
- Rego policy language
- Kubernetes-native
- Free and open source

Sentinel: Terraform's Native Guardian

Sentinel policies run automatically as part of your Terraform Cloud workflow. They examine the planned changes before any infrastructure is created or modified.

Each policy has an enforcement level that determines what happens when it fails. Advisory policies warn but allow deployment. Soft-mandatory policies require an override from someone with special permissions. Hard-mandatory policies block deployment entirely—no exceptions.

This flexibility lets you start with warnings for new policies, then gradually increase enforcement as teams adapt.

Real-World Sentinel Policy: Naming Standards

Let's say your organization requires all S3 buckets to follow the pattern: `company-environment-purpose`. Here's how Sentinel enforces this:

```
import "tfplan/v2" as tfplan

# Find all S3 buckets in the plan
s3_buckets = filter tfplan.resource_changes as _, rc {
  rc.type is "aws_s3_bucket" and
  rc.mode is "managed" and
  (rc.change.actions contains "create" or rc.change.actions contains "update")
}

# Validate naming pattern
bucket_name_valid = rule {
  all s3_buckets as _, bucket {
    bucket.change.after.bucket matches "^company-(prod|dev|staging)-[a-z0-9-]+$"
  }
}

# Enforce the rule
main = rule {
  bucket_name_valid
}
```

If someone tries to create a bucket named `my-test-bucket`, the deployment fails with a clear message explaining the required format.

Sentinel Policy: Required Tags

The Business Need

Your finance team needs to track cloud costs by department and project. Every resource must have `Department`, `Project`, and `CostCenter` tags.

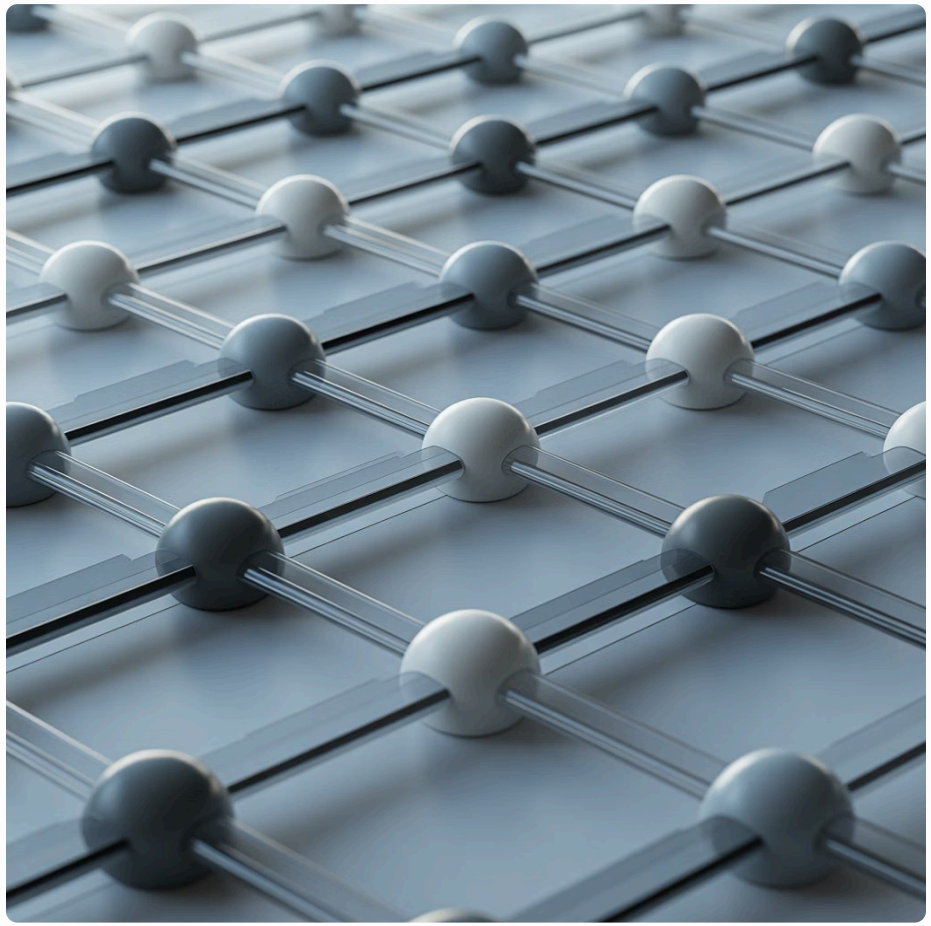
The Policy

```
required_tags = ["Department",
"Project", "CostCenter"]

main = rule {
  all aws_resources as _, r {
    all required_tags as tag {
      r.change.after.tags contains tag
    }
  }
}
```

The Result

Any resource created without all three tags triggers a deployment failure. The error message lists the missing tags, making it easy for developers to fix.

OPA: The Universal Policy Engine



Open Policy Agent uses a declarative language called Rego to define policies. While Sentinel is Terraform-specific, OPA works with Kubernetes, API gateways, microservices, and infrastructure as code.

OPA policies answer yes/no questions: "Is this deployment allowed?" "Does this configuration comply with security standards?" "Should this API request proceed?"

Many organizations use OPA for Kubernetes admission control, then extend the same policy framework to Terraform deployments for consistency.

OPA Policy Example: Resource Restrictions

Your organization prohibits expensive instance types in non-production environments. Here's an OPA policy that enforces this rule:

```
package terraform.analysis

import input as tfplan

# Prohibited instance types for dev/staging
prohibited_types := ["m5.8xlarge", "m5.16xlarge", "c5.9xlarge", "c5.18xlarge"]

# Determine environment from tags or workspace
is_production {
  tfplan.workspace == "production"
}

# Find EC2 instances
ec2_instances[resource] {
  resource := tfplan.resource_changes[_]
  resource.type == "aws_instance"
}

# Deny if non-prod uses prohibited types
deny[msg] {
  not is_production
  instance := ec2_instances[_]
  instance.change.after.instance_type == prohibited_types[_]
  msg := sprintf("Instance type %s not allowed in non-production", [instance.change.after.instance_type])
}
```

This prevents teams from accidentally deploying costly resources in development environments.

Common Compliance Rules

Most organizations implement similar baseline policies across their infrastructure. Here are patterns that apply broadly:

Tagging Standards

Require cost allocation tags like Department, Project, and Environment on all billable resources.

Naming Conventions

Enforce consistent naming patterns that identify resource purpose, environment, and ownership.

Encryption Requirements

Mandate encryption at rest for databases, storage buckets, and volumes containing sensitive data.

Network Controls

Prevent public exposure of databases and restrict security group rules to specific IP ranges.

Cost Guardrails

Block expensive instance types in development environments and require approval for large deployments.

Security Baselines

Enforce minimum security standards like HTTPS-only, IAM role usage, and private subnet placement.

Security Scanning

Detecting vulnerabilities before they reach production

Introducing Checkov

Checkov is an open-source static analysis tool that scans infrastructure code for security and compliance issues. It works with Terraform, CloudFormation, Kubernetes, and other infrastructure tools.

What Checkov Checks

- Security misconfigurations
- Compliance violations (CIS, PCI-DSS, HIPAA)
- Cloud provider best practices
- Secrets accidentally committed to code
- Overly permissive access controls

How It Works

Checkov analyzes your infrastructure code without deploying anything. It compares your configuration against a database of security policies and highlights potential issues.

Think of it as a spell-checker for security—catching problems before they become incidents.

Running Checkov: A Practical Example

Let's scan a Terraform project for security issues. Installing and running Checkov takes just a few commands:

```
# Install Checkov
pip install checkov

# Scan your Terraform directory
checkov -d /path/to/terraform

# Scan with specific framework
checkov -d . --framework terraform

# Output results to JSON
checkov -d . -o json > scan-results.json
```

Within seconds, Checkov reports any security misconfigurations, compliance violations, or best practice deviations it finds.

Understanding Checkov Results

Checkov categorizes findings by severity and provides clear remediation guidance. Each finding includes the specific resource, the violated policy, and how to fix it.

Critical: Public S3 Bucket

Issue: S3 bucket allows public read access

Risk: Sensitive data could be exposed to the internet

Fix: Add `acl = "private"` to bucket configuration

High: Unencrypted Database

Issue: RDS instance lacks encryption at rest

Risk: Data vulnerable if storage media is compromised

Fix: Set storage_encrypted = true

Checkov in Your CI/CD Pipeline

The real power of Checkov emerges when integrated into your continuous integration workflow. Security scanning becomes automatic, happening on every pull request.

1

Developer Commits Code

A developer pushes Terraform changes to a feature branch and opens a pull request.

2

CI Runs Checkov

GitHub Actions or GitLab CI automatically runs Checkov against the changed files.

3

Review and Fix

If issues are found, the build fails and the developer receives immediate feedback with remediation steps.

4

Safe Deployment

Only code that passes security scans can be merged and deployed to production.

Custom Checkov Policies

Beyond built-in checks, you can write custom policies specific to your organization's requirements. This extends Checkov to enforce internal standards.

Example: Require Backup Tags

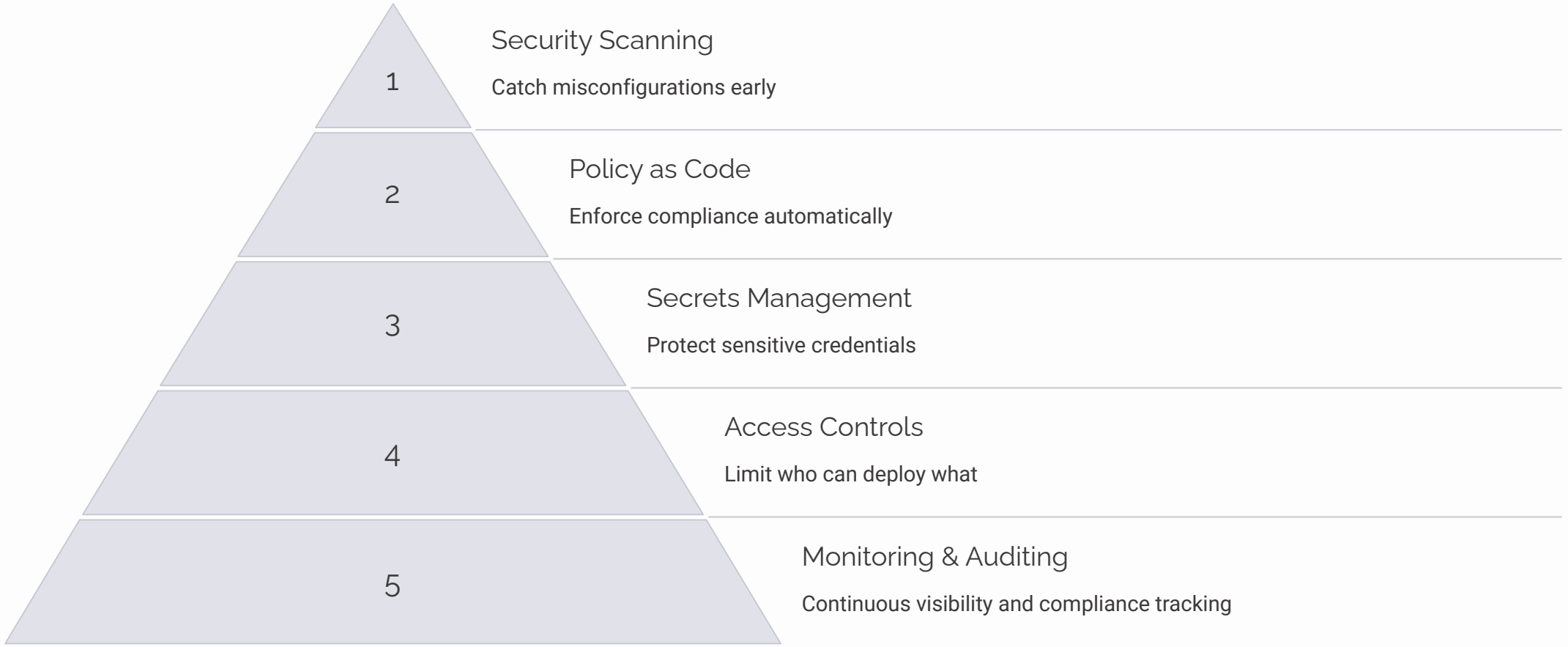
```
from checkov.common.models.enums import CheckResult
from checkov.terraform.checks.resource.base_resource_check import
BaseResourceCheck

class EnsureBackupTag(BaseResourceCheck):
    def __init__(self):
        name = "Ensure backup tag is set"
        id = "CUSTOM_001"
        supported_resources = ['aws_instance', 'aws_db_instance']
        categories = ['general']
        super().__init__(name=name, id=id, categories=categories,
supported_resources=supported_resources)

    def scan_resource_conf(self, conf):
        if 'tags' in conf:
            if 'Backup' in conf['tags'][0]:
                return CheckResult.PASSED
        return CheckResult.FAILED

check = EnsureBackupTag()
```

This custom policy ensures all EC2 instances and RDS databases have a `Backup` tag, enabling your backup automation to discover which resources need protection.

Custom policies give you flexibility to enforce standards unique to your organization, industry regulations, or security requirements.

Bringing It All Together

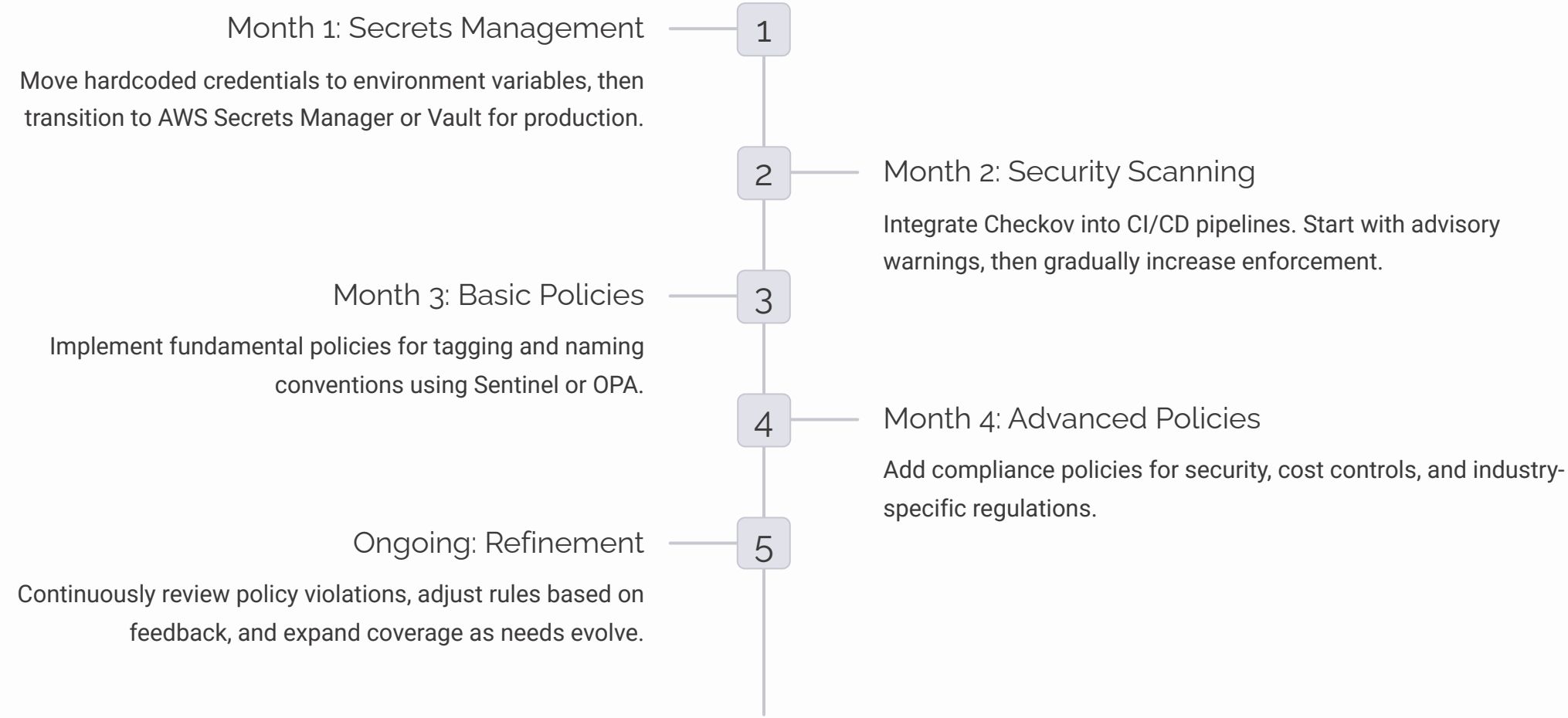
Effective infrastructure security requires multiple defensive layers working together. No single tool solves every challenge.



This defense-in-depth approach ensures that if one control fails, others catch potential issues before they become security incidents.

Implementation Roadmap

Start small and build incrementally. Trying to implement everything at once overwhelms teams and creates resistance.



Key Takeaways

1

Never Hardcode Secrets

Use environment variables for development, AWS Secrets Manager for cloud-native deployments, or Vault for enterprise needs. Always mark sensitive values in Terraform.

2

Automate Compliance

Policy as code transforms manual reviews into automated guardrails. Start with common rules like tagging and naming, then expand to security and cost controls.

3

Scan Everything

Integrate Checkov into your CI/CD pipeline to catch security issues before deployment. Custom policies extend coverage to your specific requirements.

4

Build Defense in Depth

No single tool provides complete security. Combine secrets management, policy enforcement, security scanning, and monitoring for comprehensive protection.