Securing AWS using Terrascan

Author: Vedang Joshi
Published: June 27, 2023 · 5 min read

Using Static Code Analysis to Minimize Security Risk on AWS with Terraform and Terrascan

Developing software is a complex process. With each line of code, there is potential for error. Security risks, in particular, can be devastating, leading to data breaches, system downtime, and even monetary loss. To help mitigate these risks when deploying infrastructure using Terraform on AWS, we can utilize static code analysis. A critical tool in this process is Terrascan, which allows developers to automate the process of checking their Infrastructure as Code (IaC) for potential security vulnerabilities.

What is Terrascan?

Terrascan is an open-source static code analysis tool that scans Terraform code for security vulnerabilities and compliance violations. Terrascan comes with pre-written policies that check for security best practices in your IaC code. It uses Rego, a policy-as-code language, to perform these checks.

Terrascan has extensive support for AWS and other major cloud service providers, making it a highly versatile tool for cloud deployments.

What Policies are Checked by Terrascan?

There are several crucial policies that Terrascan checks. Let's focus on the most important ones related to AWS:

1. S3 Bucket Policies: It ensures that S3 buckets are not publicly accessible, which is crucial to preventing unauthorized access to stored data.
2. IAM Policies: It checks that minimal access policies are in place for your IAM roles, to avoid granting unnecessary permissions to entities.
3. Encryption Policies: It confirms that data at rest and in transit are encrypted, offering a necessary layer of security.
4. Logging and Monitoring Policies: It validates that AWS CloudTrail and CloudWatch are enabled, to monitor and record activities for security analysis.
5. Security Group Rules: It checks for overly permissive inbound and outbound rules in security groups, which could expose your resources to potential attacks.

Integrating Terrascan as a Pre-commit Hook

Having static code analysis as a pre-commit hook is a best practice that can save you from pushing insecure code to the repository. Here are the steps to integrate Terrascan as a pre-commit hook:

Step 1: Install `pre-commit`. You can do so by running the following command:

```
pip install pre-commit
```

Step 2: Create a `.pre-commit-config.yaml` file in the root of your repository and add the following:

```
repos:
  - repo: https://github.com/tenable/terrascan
    rev: v1.8.0  # Use the ref you want to point at
    hooks:
      - id: terrascan
        args: ['-f', 'yaml', '-o', 'results']
```

This configuration specifies that the Terrascan tool will be run before every commit, outputting the results in a YAML file.

Step 3: Install the pre-commit hook. Run the following command:

```
pre-commit install
```

Now, whenever you try to commit changes, Terrascan will automatically scan your Terraform code. If any violations are found, the commit will be blocked, and you'll need to fix the problems before the commit can proceed.

Employing static code analysis with Terrascan in your Terraform code development process can significantly minimize the security risks when deploying resources on AWS. By integrating it as a pre-commit hook, you'll ensure that every line of code committed to your repository adheres to security best practices, thus providing robust security for your cloud infrastructure.

Cutsom policies

Here's how you can create custom policies in Terrascan:

Step 1: Define Conditions: Start by defining the conditions that will trigger a policy violation. These conditions are written in the Open Policy Agent's (OPA) policy-as-code language, Rego. For example, you might create a condition that checks if an AWS S3 bucket is publicly readable.

Step 2: Create a Rule: After defining your conditions, the next step is to create a rule. A rule in OPA is a named collection of conditions. The name of the rule is the policy violation that will be reported by Terrascan when the conditions of the rule are met.

Step 3: Package the Rule: The rule you've created now needs to be part of a package. A package in OPA is akin to a namespace, grouping related rules together. This makes managing your policies easier, especially when dealing with a large number of rules.

Step 4: Test the Policy: Before incorporating your custom policy into Terrascan, you should test it to make sure it works as expected. This can be done locally using OPA's `opa eval` command, which lets you evaluate your policy against sample data to verify its correctness.

Step 5: Load the Policy into Terrascan: Once you're satisfied with your custom policy, it's time to load it into Terrascan. This can be done through the `--policy` path command line argument when you run Terrascan. Alternatively, you can place your custom policies into a directory and set the `POLICY_PATH` environment variable to the directory's path.

By following these steps, you can extend the capabilities of Terrascan with your custom policies, enabling more robust security checks that cater specifically to your organization's requirements.

Some usecases that require custom policies

Here are some scenarios where you might need to create custom policies:

1. Validating the existence of DDoS prevention mechanisms, such as by assessing the request per second (RPS) limit on Kubernetes ingress.
2. Ensuring that newly created resources adhere to appropriate tagging and naming conventions.

In conclusion, utilizing Terrascan's static code analysis for your Terraform AWS deployments, along with the ability to craft custom policies, brings robust and tailored security to your cloud infrastructure. It's an indispensable tool for automating security best practices and ensuring compliance in your infrastructure-as-code development process. As always, Thanks for reading!

Security Terraform DevOps