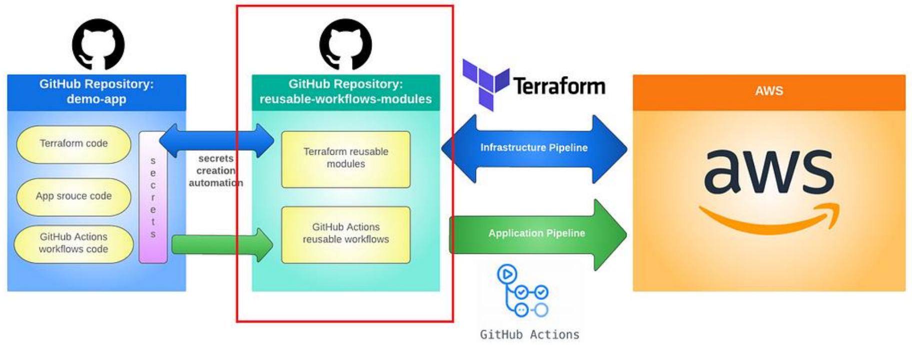
DevOps Self-Service Centric Pipeline Security and Guardrails

Coming from a traditional DevOps mindset, you may still be a bit skeptical. What about security measures and guardrails for such DevOps self-service? Security is at the forefront of everyone's mind nowadays. Designing and implementing your pipelines to ensure sound security is paramount.

DevOps self-service calls for relinquishing control to the developers. How can we have peace of mind when handing DevOps pipeline ownership to developers? Great question you asked. We hear you loud and clear! Security and guardrails are key implementation pieces on this path to DevOps self-service. In this article, let's focus on pipeline security and guardrails, and you will see why you can have peace of mind when rolling out your DevOps self-service practice.

First, let's briefly revisit our pipeline architecture. Notice the part highlighted in red below: This is the repository where your reusable Terraform modules and GitHub Actions workflows reside. Here, you'll need to implement your pipeline security and guardrails. This repository hosting your reusable workflows and modules acts like the engine of your DevOps self-service vehicle. With these security and guardrail measures implemented in this centralized repository, you have a higher quality engine, which helps guarantee a smoother ride.

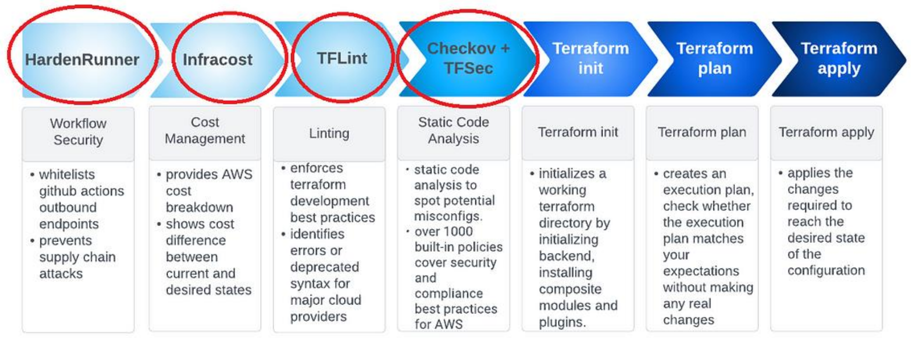


Developers, when implementing their pipeline logic in the caller workflows from their applications, do not need to know about the details of the security and guardrail measures. They are the consumers of the finished product of your high-quality reusable workflows and modules, and they automatically catch any security or guardrail issues developers introduce.

The following list of tools, highlighted in red in the diagrams of the sample infrastructure pipeline and sample application pipelines below, have been hand-picked to improve the security posture of your overall platform:

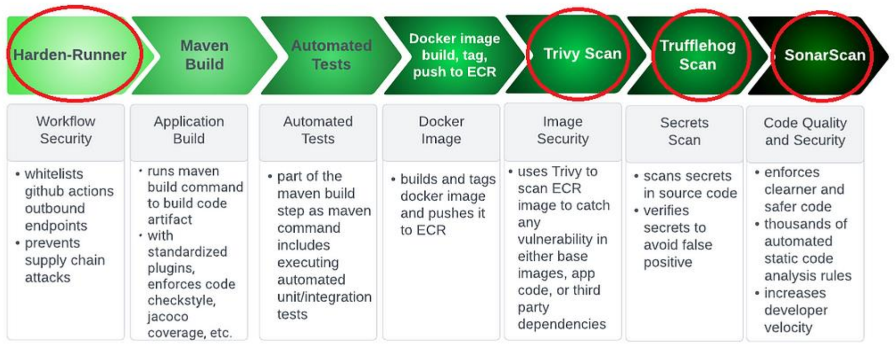
- your pipelines
- your infrastructure
- your source code
- your base images

- your dependent libraries



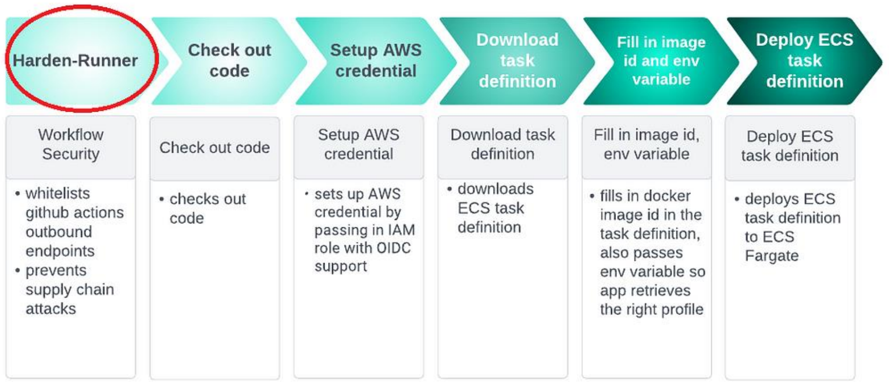
HardenRunner	Infracost	TFLint	Checkov + TFSec	Terraform init	Terraform plan	Terraform apply
Workflow Security	Cost Management	Linting	Static Code Analysis	Terraform init	Terraform plan	Terraform apply
• whitelists github actions outbound endpoints • prevents supply chain attacks	• provides AWS cost breakdown • shows cost difference between current and desired states	• enforces terraform development best practices • identifies errors or deprecated syntax for major cloud providers	• static code analysis to spot potential misconfigs. • over 1000 built-in policies cover security and compliance best practices for AWS	• initializes a working terraform directory by initializing backend, installing composite modules and plugins.	• creates an execution plan, check whether the execution plan matches your expectations without making any real changes	• applies the changes required to reach the desired state of the configuration

Sample infrastructure pipeline.



Harden-Runner	Maven Build	Automated Tests	Docker image build, tag, push to ECR	Trivy Scan	Trufflehog Scan	SonarScan
Workflow Security	Application Build	Automated Tests	Docker Image	Image Security	Secrets Scan	Code Quality and Security
• whitelists github actions outbound endpoints • prevents supply chain attacks	• runs maven build command to build code artifact • with standardized plugins, enforces code checkstyle, jacoco coverage, etc.	• part of the maven build step as maven command includes executing automated unit/integration tests	• builds and tags docker image and pushes it to ECR	• uses Trivy to scan ECR image to catch any vulnerability in either base images, app code, or third party dependencies	• scans secrets in source code • verifies secrets to avoid false positive	• enforces clearner and safer code • thousands of automated static code analysis rules • increases developer velocity

Sample application pipeline (CI workflow).



Harden-Runner	Check out code	Setup AWS credential	Download task definition	Fill in image id and env variable	Deploy ECS task definition
Workflow Security	Check out code	Setup AWS credential	Download task definition	Fill in image id, env variable	Deploy ECS task definition
• whitelists github actions outbound endpoints • prevents supply chain attacks	• checks out code	• sets up AWS credential by passing in IAM role with OIDC support	• downloads ECS task definition	• fills in docker image id in the task definition, also passes env variable so app retrieves the right profile	• deploys ECS task definition to ECS Fargate

Sample application pipeline (CD workflow).

Let’s dive in and take a closer look at each highlighted action.

Harden-Runner

Harden-Runner is the only action used in all pipelines — infrastructure and application pipelines for CI and CD. Why? Because of the unique nature and purpose of Harden-Runner.

Developed by StepSecurity, Harden-Runner is a purpose-built security monitoring agent for your pipelines to detect and prevent malicious patterns observed during past software supply chain security breaches. The main features of Harden-Runner include:

- Automatically discovers and correlates outbound traffic with each step in your pipeline, as compromised dependencies and build tools typically make outbound calls.
- Prevent exfiltration of credentials in the pipeline.
- Detect tampering of source code during the build.
- Detect compromised dependencies and build tools.

Harden-Runner is configured with two steps. First, you must enable its `audit` mode. Here's how to do that:

```
- name: Harden Runner
  uses: step-security/harden-runner@ebacdc22ef6c2cfb85ee5ded8f2e640f4c776dd5
  with:
    egress-policy: audit # TODO: change to 'egress-policy: block' after
couple of runs
```

After running a couple of workflow runs, you can enable its `block` mode by whitelisting the outbound endpoints for all the actions called in your workflow. You can also configure notifications to be sent out via email or Slack if outbound endpoints that are not in the `allowed-endpoints` list are called. The sample snippet below will help you see how this works:

```
- name: Harden Runner
  uses: step-security/harden-runner@ebacdc22ef6c2cfb85ee5ded8f2e640f4c776dd5
  with:
    disable-sudo: true
    egress-policy: block
    allowed-endpoints: >
      api.adoptopenjdk.net:443
      api.ecr.us-east-1.amazonaws.com:443
      apk.corretto.aws:443
      dl-cdn.alpinelinux.org:443
      ghcr.io:443
      github.com:443
      objects.githubusercontent.com:443
```

Also, note the line `disable-sudo: true` in the above snippet. `Sudo` allows the user to delegate privileges to run commands as a root or another user. This drastically increases the risk because the root user can do anything to the system. Harden-Runner monitors root usage in audit mode and makes these findings available on the insights page. If Harden-Runner doesn't observe any `sudo` calls, you will see a recommendation to set `disable-sudo` to true. It's a security best practice to disable `sudo` if you are not using it.

With Harden-Runner, you are confident this shield protects your pipeline well. It's fair to say that Harden-Runner is your pipelines' security guard.

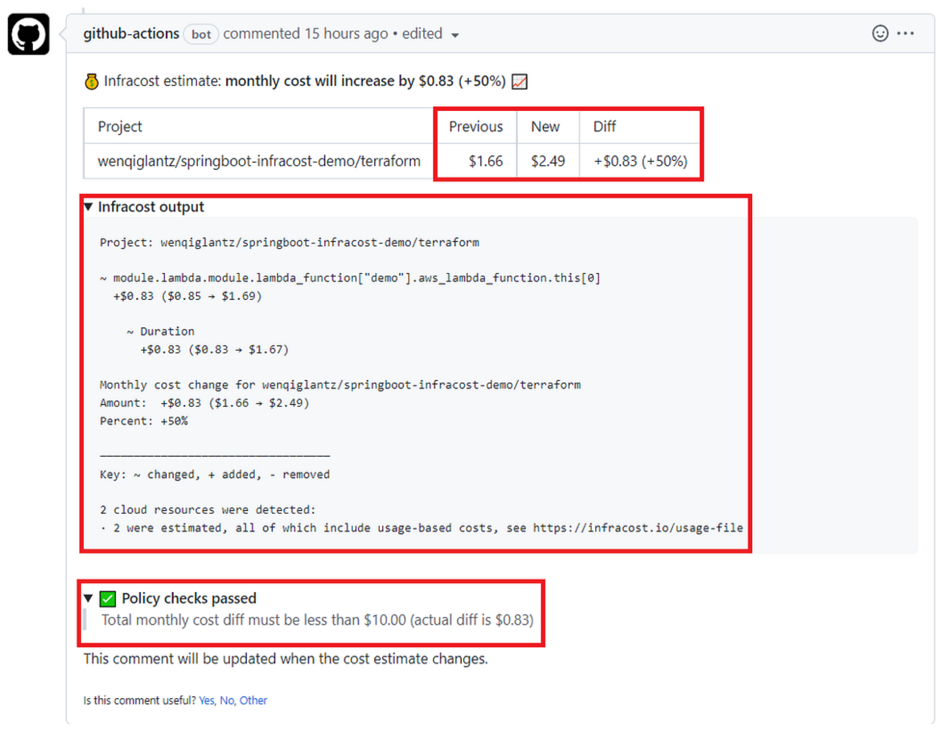
StepSecurity maintains great documentation on Harden-Runner. Step-by-step guidance can be found at https://docs.stepsecurity.io/ . Do check it out!

I wrote an article a few months ago on A First Look at Harden Runner: the Must Have GitHub Action to Prevent Supply Chain Attacks. Feel free to check it out for more details.

Infracost

So often, spikes in cloud cost get captured after you have incurred the cost. Infracost lets DevOps, SRE, and engineers see a cost breakdown and understand costs before making changes in the terminal or pull requests. This provides your team with a safety net to catch abnormal cloud cost estimates due to fat fingering or misconfiguration in Terraform configuration.

See the following screenshot of a sample pull request (PR) comment added by Infracost when you create a new PR with Terraform code changes for your application. The diff amount between the previous cost and the new cost based on the latest Terraform code changes is $0.83. Small as it may be, the diff feature enables you to validate your Terraform code changes against a predefined diff threshold. Your workflow will fail if the diff amount exceeds the predefined threshold amount. This indeed serves as the guardrail for your cloud cost management while opening up the infrastructure code control to the developers.



github-actions bot commented 15 hours ago • edited

💰 Infracost estimate: **monthly cost will increase by $0.83 (+50%)** 📈

Project	Previous	New	Diff
wenqiglantz/springboot-infracost-demo/terraform	$1.66	$2.49	+$0.83 (+50%)

Infracost output

```
Project: wenqiglantz/springboot-infracost-demo/terraform

~ module.lambda.module.lambda_function["demo"].aws_lambda_function.this[0]
  +$0.83 ($0.85 → $1.69)

    ~ Duration
      +$0.83 ($0.83 → $1.67)

Monthly cost change for wenqiglantz/springboot-infracost-demo/terraform
Amount:  +$0.83 ($1.66 → $2.49)
Percent: +50%

──────────────────────────────────
Key: ~ changed, + added, - removed

2 cloud resources were detected:
∙ 2 were estimated, all of which include usage-based costs, see https://infracost.io/usage-file
```

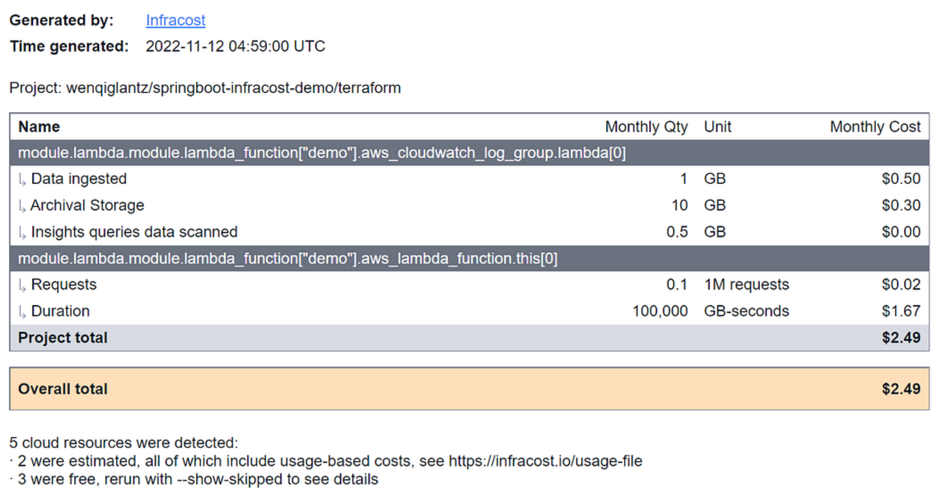
✅ Policy checks passed

Total monthly cost diff must be less than $10.00 (actual diff is $0.83)

This comment will be updated when the cost estimate changes.

Is this comment useful? Yes, No, Other

The next screenshot shows a sample email of a customized Infracost workflow that can notify you of your application's monthly costs. These costs are based on your static Terraform configuration and the dynamic usage configuration for your cloud resources.



Generated by:	Infracost
Time generated:	2022-11-12 04:59:00 UTC

Project: wenqiglantz/springboot-infracost-demo/terraform

Name	Monthly Qty	Unit	Monthly Cost
module.lambda.module.lambda_function["demo"].aws_cloudwatch_log_group.lambda[0]			
└ Data ingested	1	GB	$0.50
└ Archival Storage	10	GB	$0.30
└ Insights queries data scanned	0.5	GB	$0.00
module.lambda.module.lambda_function["demo"].aws_lambda_function.this[0]			
└ Requests	0.1	1M requests	$0.02
└ Duration	100,000	GB-seconds	$1.67
Project total			**$2.49**

Overall total	**$2.49**

5 cloud resources were detected:
∙ 2 were estimated, all of which include usage-based costs, see https://infracost.io/usage-file
∙ 3 were free, rerun with --show-skipped to see details

Here are some of the key steps you can include in your infracost GitHub Actions workflow, notice the comments above each step, which walk you through how you arrive at the diff amount on your infrastructure cost based on your base branch and your PR branch, and a report can be generated, also a comment can be added to your PR to indicate the diff details, as seen in the screenshot above.

```
    # this step calls infracost/actions/setup@v2, which installs the latest
patch version of the Infracost CLI v0.10.x and
```

```
    # gets the backward-compatible bug fixes and new resources. Replacing the
version number with git SHA is a security hardening measure.
    - name: Setup Infracost
      uses: infracost/actions/setup@6bdd3cb01a306596e8a614e62af7a9c0a133bc5c
      with:
        api-key: ${{ secrets.INFRACOST_API_KEY }}

    # Checkout the base branch of the pull request (e.g. main).
    - name: Checkout base branch
      uses: actions/checkout@93ea575cb5d8a053eaa0ac8fa3b40d7e05a33cc8
      with:
        ref: '${{ github.event.pull_request.base.ref }}'

    # Generate Infracost JSON file as the baseline.
    - name: Generate Infracost cost estimate baseline
      run: |
        export INFRACOST_API_KEY=${{ secrets.INFRACOST_API_KEY }}
        cd ${TF_ROOT}
        infracost breakdown --path=. \
                            --terraform-var-file=${{ inputs.terraform-var-
file }} \
                            --usage-file ${{ inputs.usage-file }} \
                            --format=json \
                            --out-file=/tmp/infracost-base.json

    # Checkout the current PR branch so we can create a diff.
    - name: Checkout PR branch
      uses: actions/checkout@93ea575cb5d8a053eaa0ac8fa3b40d7e05a33cc8

    # Generate an Infracost diff and save it to a JSON file.
    - name: Generate Infracost diff
      run: |
        export INFRACOST_API_KEY=${{ secrets.INFRACOST_API_KEY }}
        cd ${TF_ROOT}
        infracost diff --path=. \
                        --format=json \
                        --show-skipped \
                        --terraform-var-file=${{ inputs.terraform-var-file }}
\
                        --usage-file ${{ inputs.usage-file }} \
                        --compare-to=/tmp/infracost-base.json \
                        --out-file=/tmp/infracost.json

    # generate the html report based on the JSON output from last step
    - name: Generate Infracost Report
      run: |
        export INFRACOST_API_KEY=${{ secrets.INFRACOST_API_KEY }}
        cd ${TF_ROOT}
        infracost output --path /tmp/infracost.json --show-skipped --format
html --out-file report.html

    # upload the report to artifact so subsequent workflow can download the
report and email it as attachment
    - uses: actions/upload-artifact@0b7f8abb1508181956e8e162db84b466c27e18ce
# v3.1.2
      with:
        name: report.html
        path: ${{ inputs.working-directory }}/report.html

    # Posts a comment to the PR using the 'update' behavior.
    # This creates a single comment and updates it. The "quietest" option.
    # The other valid behaviors are:
    #   delete-and-new - Delete previous comments and create a new one.
    #   hide-and-new - Minimize previous comments and create a new one.
    #   new - Create a new cost estimate comment on every push.
    #   update - Update a cost estimate comment when there is a change in the
cost estimate.
    # See https://www.infracost.io/docs/features/cli_commands/#comment-on-
pull-requests for other options.
    - name: Post Infracost comment
      run: |
        export INFRACOST_API_KEY=${{ secrets.INFRACOST_API_KEY }}
        infracost comment github --path=/tmp/infracost.json \
                                 --repo=$GITHUB_REPOSITORY \
                                 --github-token=${{github.token}} \
                                 --pull-
request=${{github.event.pull_request.number}} \
                                 --behavior=update \
                                 --policy-path=${TF_ROOT}/infracost-
policy.rego
```

Infracost is a must-have tool in rolling out your DevOps self-service practice. In addition to the open-source version, which can be integrated into your infrastructure pipeline, Infracost offers a paid cloud version with many cool features. Some of these perks are a centralized dashboard, which gives you a glance at all your PRs and the cost changes they introduce, integration into issue trackers such as JIRA, guardrails that help you control costs by monitoring PRs, and the ability to trigger actions when your defined thresholds are exceeded.

One recently introduced feature in Infracost Cloud is its centralized cost policies. These let you define central policies and scan your repositories for cost-saving opportunities. How cool is that! For more details, refer to Centralized cost policies | Infracost.

Infracost maintains excellent documentation and detailed instructions to get you started. You can find out more at https://www.infracost.io/docs/ . Infracost is indeed a rare gem in the open source space. Definitely add it to your toolkit for your DevOps self-service.

I wrote an article with step-by-step instructions on integrating Infracost into your Terraform workflow a few months ago. Feel free to check it out: Infracost + Terraform + GitHub Actions = Automate Cloud Cost Management.

TFLint

TFLint is a framework, and each feature is provided by plugins. The key features are as follows:

- Find possible errors (like invalid instance types).
- Warn about deprecated syntax and unused declarations.

- Enforce best practices and naming conventions.

Incorporate TFLint in Terraform GitHub Actions workflow for your project to automate linting during Terraform workflow execution. See some sample actions/steps for TFLint in an infrastructure pipeline in the example below:

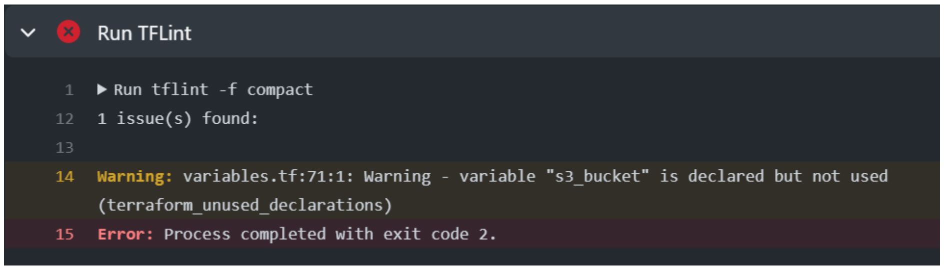
```
- uses: actions/cache@937d24475381cd9c75ae6db12cb4e79714b926ed # v2.1.7
  name: Cache plugin dir
  with:
    path: ~/.tflint.d/plugins
    key: ubuntu-latest-tflint-${{ hashFiles('.tflint.hcl') }}

- uses: terraform-linters/setup-
tflint@444635365d380c7363d1eaee4267317c2445717d # v2.0.1
  name: Setup TFLint
  with:
    tflint_version: latest

- name: Init TFLint
  run: tflint --init

- name: Run TFLint
  run: tflint -f compact
```

Also, here's a screenshot of a sample TFLint error captured from the Terraform GitHub Actions workflow. As you can see, there are warning points for the variable `s3_bucket`, which is declared but not used. TFLint really helps keep your code clean and concise. With standards and best practices enforced by frameworks/tools like TFLint, DevOps self-service can be a breeze!



```
▶ Run tflint -f compact
1 issue(s) found:

Warning: variables.tf:71:1: Warning - variable "s3_bucket" is declared but not used
(terraform_unused_declarations)
Error: Process completed with exit code 2.
```

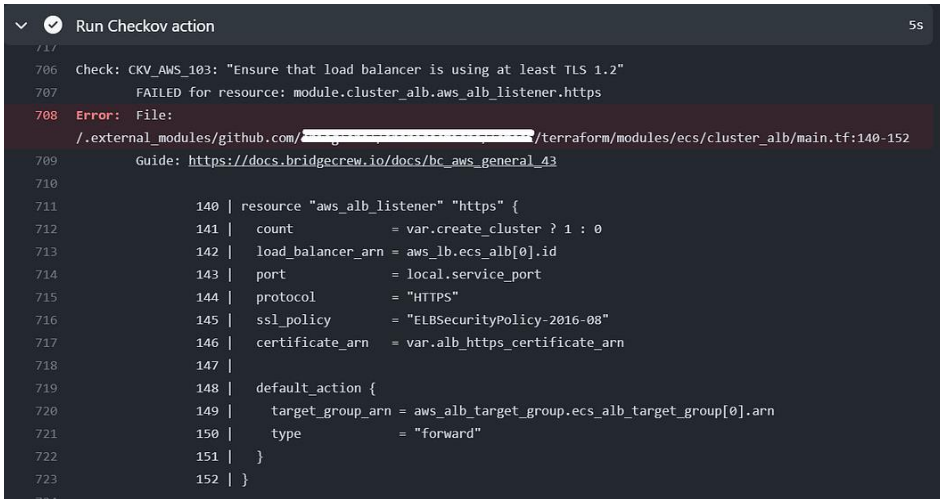
Checkov

Checkov is a static code analysis tool for infrastructure as code (IaC) and software composition analysis (SCA) for images and open source packages. With over 1,000 built-in policies covering security and compliance best practices for all major cloud providers, Checkov scans Terraform, Terraform Plan, CloudFormation, AWS SAM, Kubernetes, Dockerfile, Serverless framework, Bicep, and ARM template files. It also supports in-line suppression of accepted risks or false positives to reduce recurring scan failures.

The following step for Checkov action can be incorporated into your Terraform GitHub Actions workflow:

```
- name: Run Checkov action
  uses: bridgecrewio/checkov-action@3854b91536303a096e7693434ef98706a0be82cb
# master
  with:
    directory: ${{ inputs.working-directory }}
    quiet: true # optional: display only failed checks
    soft_fail: true # optional: do not return an error code if there are
failed checks
    framework: terraform # optional: run only on a specific infrastructure
{cloudformation,terraform,kubernetes,all}
    output_format: sarif # optional: the output format, one of: cli, json,
junitxml, github_failed_only, or sarif. Default: sarif
    output_file_path: reports/results.sarif # folder and name of results file
    download_external_modules: true # optional: download external terraform
modules from public git repositories and terraform registry
    log_level: DEBUG # optional: set log level. Default WARNING
```

Here's a sample failed check of Checkov. It includes the description of the point, file name, line numbers, and remediation guide documentation links:


Run Checkov action 5s

```
Check: CKV_AWS_103: "Ensure that load balancer is using at least TLS 1.2"
        FAILED for resource: module.cluster_alb.aws_alb_listener.https
Error:  File:
/.external_modules/github.com/[redacted]/terraform/modules/ecs/cluster_alb/main.tf:140-152
        Guide: https://docs.bridgecrew.io/docs/bc_aws_general_43

                140 | resource "aws_alb_listener" "https" {
                141 |   count             = var.create_cluster ? 1 : 0
                142 |   load_balancer_arn = aws_lb.ecs_alb[0].id
                143 |   port              = local.service_port
                144 |   protocol          = "HTTPS"
                145 |   ssl_policy        = "ELBSecurityPolicy-2016-08"
                146 |   certificate_arn   = var.alb_https_certificate_arn
                147 |
                148 |   default_action {
                149 |     target_group_arn = aws_alb_target_group.ecs_alb_target_group[0].arn
                150 |     type             = "forward"
                151 |   }
                152 | }
```

More information on Checkov can be found on its GitHub repository https://github.com/bridgecrewio/checkov .

TFSec

Recommended by ThoughtWorks, TFSec uses static analysis of the Terraform code to spot potential misconfigurations. Here are the main features of TFSec:

- Checks for misconfigurations across all major (and some minor) cloud providers
- Hundreds of built-in rules
- Evaluates HCL expressions as well as literal values
- Evaluates Terraform functions, e.g., `concat()`
- Evaluates relationships between Terraform resources
- Applies (and embellishes) user-defined Rego policies

- Supports multiple output formats: lovely (default), JSON, SARIF, CSV, CheckStyle, JUnit, text, Gif.
- Very fast. Capable of quickly scanning huge repositories

Here's a sample result point. It highlights the severity of the issue, the description, the file, the line number, and the links to more information about why TFSec flagged that line as a security issue. See more below:

```
Result #1 CRITICAL Listener uses an outdated TLS policy.
──────────────────────────────────────────────────────────────────────────
───
  github.com\###\###\terraform\modules\ecs\cluster_alb\main.tf:105
   via main.tf:64-83 (module.cluster_alb)
──────────────────────────────────────────────────────────────────────────
───
  100    resource "aws_alb_listener" "https" {
  ...
  105  [   ssl_policy        = "ELBSecurityPolicy-2016-08"
("ELBSecurityPolicy-2016-08")
  ...
  112    }
──────────────────────────────────────────────────────────────────────────
───
          ID aws-elb-use-secure-tls-policy
      Impact The SSL policy is outdated and has known vulnerabilities
  Resolution Use a more recent TLS/SSL policy for the load balancer

  More Information
  - https://aquasecurity.github.io/tfsec/v1.22.1/checks/aws/elb/use-secure-
tls-policy/
  -
https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/l
b_listener
──────────────────────────────────────────────────────────────────────────
───
```

TFSec can be incorporated into your Terraform GitHub Actions workflow, so when you provision infrastructure for your app, TFSec will start by doing a static Terraform code analysis to detect potential security risks. The TFSec step in Terraform GitHub Actions workflow looks like this:

```
- name: Run tfsec, static analysis tool to detect potential security risks
  uses: aquasecurity/tfsec-pr-commenter-
action@7a44c5dcde5dfab737363e391800629e27b6376b # v1.3.1
  with:
    tfsec_args: --soft-fail
    github_token: ${{ github.token }}
```

`--soft-fail` is a flag to tell your GitHub Actions workflow to not fail the workflow if security risks are found. This should be used cautiously and only when multiple developers are working on the same application, but doing different tasks. Some may be looking into the vulnerability fix, while others actively work on feature development. Introducing the `--soft-fail` flag allows the developers to continue developing new features while the other developer(s) can work on fixing the vulnerabilities found.

There is a known issue with TFSec not working well with pinned Terraform reusable modules. I opened this issue in November 2022. The TFSec team is still working on the fix as of this writing. I am looking forward to the fix.

For more details on this action, refer to https://github.com/aquasecurity/tfsec-pr-commenter-action .

Trivy

Trivy is a comprehensive and versatile open source security scanner. It is designed to be fast and easy to use, and it can be run from the command line or integrated into a CI/CD workflow. The snippet below shows the Trivy scan step in a GitHub Actions CI workflow:

```
- name: Scan docker image with Trivy vulnerability scanner
  uses: aquasecurity/trivy-action@9ab158e8597f3b310480b9a69402b419bc03dbd5
  with:
    image-ref: ${{ steps.build-image.outputs.image }}
    format: 'table'
    exit-code: '1'
    ignore-unfixed: true
    vuln-type: 'os,library'
    severity: 'CRITICAL,HIGH'
```

Trivy scans container images for known vulnerabilities in the operating system packages and libraries that they use. It uses a database of vulnerability information provided by the National Vulnerability Database (NVD) and other sources.

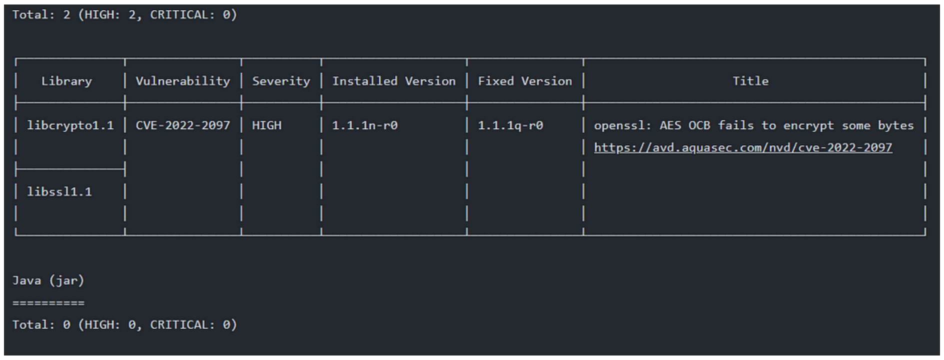
What Trivy can scan

- Container Image
- File system
- Git Repository (remote)
- Virtual Machine Image
- Kubernetes
- AWS

What Trivy can find

- OS packages and software dependencies in use (SBOM)
- Known vulnerabilities (CVEs)
- IaC issues and misconfigurations
- Sensitive information and secrets
- Software licenses

See the following screenshot of a sample Trivy scan result in the application CI workflow. The error indicates there is a vulnerability in the base image.



```
Total: 2 (HIGH: 2, CRITICAL: 0)

┌──────────────┬────────────────┬──────────┬───────────────────┬───────────────┬──────────────────────────────────────────┐
│   Library    │ Vulnerability  │ Severity │ Installed Version │ Fixed Version │                  Title                   │
├──────────────┼────────────────┼──────────┼───────────────────┼───────────────┼──────────────────────────────────────────┤
│ libcrypto1.1 │ CVE-2022-2097  │ HIGH     │ 1.1.1n-r0         │ 1.1.1q-r0     │ openssl: AES OCB fails to encrypt some bytes │
│              │                │          │                   │               │ https://avd.aquasec.com/nvd/cve-2022-2097 │
├──────────────┤                │          │                   │               │                                          │
│ libssl1.1    │                │          │                   │               │                                          │
│              │                │          │                   │               │                                          │
└──────────────┴────────────────┴──────────┴───────────────────┴───────────────┴──────────────────────────────────────────┘

Java (jar)
==========
Total: 0 (HIGH: 0, CRITICAL: 0)
```

You will need to provide a fix to update the vulnerable packages to the latest versions in the base image in your `Dockerfile`, especially under the following conditions:

- If your app uses an alpine base image, add the line `RUN apk update && apk upgrade -U -a` under the `FROM <base image>` line in your `Dockerfile`. For example:

```
FROM amazoncorretto:17-alpine-jdk
RUN apk update && apk upgrade -U -a
...
```

- If your app doesn't use alpine base image, you can add the line `RUN apt-get update && apt-get upgrade -y` under the `FROM <base image>` line in your `Dockerfile`. For example:

```
FROM openjdk:17.0.2-jdk-slim-bullseye
RUN apt-get update && apt-get upgrade -y
...
```

With the above fix, your CI should be able to pass the base image scan.

For those who have been using Trivy in your pipeline, you may have noticed the recent Trivy timeout error: "`FATAL image scan error: scan error: scan failed: failed analysis: analyze error: timeout: context deadline exceeded`". Per this issue in Trivy's GitHub repo issues, the root cause is with search.maven.org timing out intermittently. As of 1/22/2023, it was reported on this link that the intermittent timeout errors from http://search.maven.org have been resolved, and they are investigating the root cause.

TruffleHog

TruffleHog is a tool that searches through Git repositories for high entropy strings, such as passwords and API keys. It can help identify and prevent data leaks by detecting secrets accidentally committed to a repository. It works by calculating the entropy of a string, which measures the amount of randomness or complexity in the string. Strings with high entropy are more likely to be secrets, and TruffleHog searches through a repository's commit history looking for these high-entropy strings.

TruffleHog, incorporated into your application's CI workflow, can auto-detect secrets leak during CI build. Here's how to get started:

```
- name: TruffleHog Secrets Scan
  uses: trufflesecurity/trufflehog@main
  with:
    path: ./
    base: ${{ github.event.repository.default_branch }}
    head: HEAD
    extra_args: --debug --only-verified
```

The `--only-verified` flag in the `extra_args` input parameter scans verified secrets, which means live secrets can be used to log into your provider service or application. These may be your cloud provider, RDS, or any other services/applications your secrets are used for authentication. These live secrets, once leaked into the hands of bad actors, are bound to cause damage to your platform. So be sure to include TruffleHog in your pipelines. Be the police of your own application.

For more details on TruffleHog, refer to its GitHub repository https://github.com/trufflesecurity/trufflehog .

SonarScan

SonarScan is a code-scanning tool that analyzes source code for security vulnerabilities, code smells, and other issues. It is typically used in software development to help improve the quality and security of code. SonarScan can be run on various programming languages and integrates with GitHub Actions.

In addition to security vulnerabilities and code smells, SonarScan can also be configured to check for compliance with coding standards and best practices and to detect duplication and other problems in the codebase.

To add SonarScan to your GitHub Actions CI workflow, add the following action:

```
- name: SonarQube Scan
  uses: sonarsource/sonarqube-scan-action@master
  env:
    SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
    SONAR_HOST_URL: ${{ secrets.SONAR_HOST_URL }}
```

Refer to https://github.com/SonarSource/sonarqube-scan-action for more details on the usage variations of this action.

Notice we have Trivy scan, Trufflehog scan, and SonarScan triggered after the Docker image has been pushed to the container registry, such as ECR, in the application CI workflow. The design intention behind it is similar to the `--soft-fail` flag in the TFSec section. This allows multiple developers to work on the same application without being blocked for security vulnerability issues as other developer(s) work on the vulnerability fix. This is a mere recommendation. You can adjust the sequence of those security actions and guardrails based on your pipeline needs.

Summary

One major concern of DevOps self-service is how you ensure that pipeline security and guardrails are properly addressed and implemented. We looked closely at a list of hand-picked actions for pipeline security and guardrails for not only source code and dependent libraries, but also base image, infrastructure, and pipelines. These actions are implemented in the GitHub Actions reusable workflows for infrastructure and application pipelines to ensure they are adhered to by developers when developing caller workflows for their applications. I hope you find this article helpful.

References

- https://github.com/step-security/harden-runner
- https://docs.stepsecurity.io/
- Disable Sudo | StepSecurity
- https://www.infracost.io/
- https://www.infracost.io/docs/infracost_cloud/cost_policies/
- aquasecurity/tfsec: Security scanner for your Terraform code (github.com)
- GitHub — bridgecrewio/checkov: Prevent cloud misconfigurations and find vulnerabilities during build-time in infrastructure as code, container images and open source packages with Checkov by Bridgecrew
- https://github.com/aquasecurity/tfsec-pr-commenter-action
- https://github.com/aquasecurity/trivy
- https://github.com/trufflesecurity/trufflehog
- https://github.com/SonarSource/sonarqube-scan-action