October 7, 2021 () 10 mins read **Editor's note** This blog originally appeared on fugue.co. Fugue joined Snyk in 2022 and is a key component

of **Snyk IaC**. Last week we announced Fugue IaC, which enables cloud engineering teams to secure their infrastructure as code (IaC) and cloud runtime environment using the same policies. For running IaC checks locally, Fugue developed Regula, an open source tool built on Open Policy Agent (OPA). Regula itself can be used with or without Fugue. It comes with hundreds of out-of-the-box rules designed to check Terraform and CloudFormation infrastructure as code (IaC) and Kubernetes manifests. These rules are mapped to

EC2 instances to use approved AMIs. In situations like this, you can write a custom rule using OPA's Rego language, and Regula will apply it to your IaC. That way, if your IaC declares a non-approved AMI, the Regula check will fail. In this blog post: We'll write a custom rule to check AWS EC2 AMIs declared in Terraform, explaining the Rego code line by line. We'll use our open source tool Regula to test the rule out against a noncompliant Terraform file. We'll fix the noncompliant Terraform.

But sometimes there are organization-specific policies you want to enforce – for instance, requiring **Amazon**

the CIS Benchmarks and cover many security vulnerabilities.

Note that while this blog post uses Regula to check Terraform HCL, you can also use the same exact custom rule with the Fugue SaaS to check cloud runtime resources. This is thanks to Fugue's Unified Policy Engine, which supports using the same rules at IaC and runtime. You'll find all the code in this GitHub gist.

Let's go! **Getting started**

Download code files

ami.tf – a Terraform HCL file declaring two EC2 instances, one compliant and one noncompliant

In our **GitHub gist**, select the Download ZIP button and then extract the files. There are two files: approved ami.rego – a custom rule written in Rego

The next step is to install Regula, if you haven't yet. Homebrew users can execute the following commands in their terminal:

Install Regula

brew install regula

brew tap fugue/regula

You can alternatively install a <u>prebuilt binary for your platform</u> or, if you prefer, <u>run Regula with Docker</u>.

involving multiple resources or checking for missing resources.

Writing the custom rule

First, we'll determine what type of rule to write. With Regula, there are two rule types: simple and advanced. Simple

rules are useful when you're checking a single resource type. Advanced rules are better for more complex logic

We're just going to check one Terraform resource type, aws_instance, so we'll write a simple rule.

Let's start with the basics. All Regula rules must have a package declaration beginning with rules. and ending with a short identifier (<u>line 4</u>):

Metadata

10

11

12

13

14

15

16

Package declaration

4 package rules.approved_ami

We can optionally add some metadata (lines 6-18):

"id": "CUSTOM_0002",

6 __rego__metadoc__ := {

"controls": {

"severity": "High"

20 resource_type = "aws_instance"

"custom": {

},

17 18 }

This isn't required, but it makes Regula's report output even more informative.

Next up, we state which **Terraform resource type** we want to check (**line 20**):

"CORPORATE-POLICY": [

"CORPORATE-POLICY_1.2"

"title": "AWS EC2 instances must use approved AMIs",

"description": "Per company policy, EC2 instances may only use AMI IDs from a

This syntax means that the input will be a single AWS EC2 instance. When Regula applies the rule to our Terraform, it

will evaluate a single instance at a time. **Approved AMI set**

ami-09e67e426f25ce0d7

ami-03d5c68bab01f3496

22 approved_amis = {

Resource type

(If you're curious, these are the IDs for Ubuntu Server 20.04 LTS HVM, SSD Volume Type in us-east-1 and us-west-2,

"ami-09e67e426f25ce0d7", # us-east-1

Now, we'll create a **set** containing the approved AMI IDs. We'll call it **approved_amis** (**lines 22-26**):

Ubuntu Server 20.04 LTS (HVM), SSD Volume Type

For simplicity's sake, let's say your organization has blessed these two AMI IDs:

25 "ami-03d5c68bab01f3496" # us-west-2 26 }

The deny rule

resource should fail the check.

respectively.)

24

We're going to get a little fancy here and return a <u>custom error message</u> that lists the unapproved AMI ID when an instance fails the Regula check.

What our deny rule should do is check whether the currently evaluated instance's AMI ID is in the approved set, and if

it isn't, deny should return an error message for that resource (i.e., produce a failing rule result). deny is defined

msg = sprintf("%s is not an approved AMI ID", [input.ami])

Finally, we'll write a deny rule. This is where the main logic lives! The deny rule defines the conditions in which a

Let's take a closer look at the logic: 29 not approved_amis[input.ami]

28 deny[msg] {

not approved_amis[input.ami]

in <u>lines 28-31</u>:

31 }

the set.

error message."

We define the error message in this line:

We see the following output:

[1]: aws_instance.bad

See a more detailed report

"rule_results": [

"controls": [

"CORPORATE-POLICY_1.2"

"rule_id": "CUSTOM_0002",

"rule_result": "FAIL",

"source_location": [

"line": 13,

"column": 1

"controls": [

"rule_severity": "High",

"path": "ami.tf",

"CORPORATE-POLICY_1.2"

"filepath": "ami.tf",

"input_type": "tf",

"source_location": [

"line": 8,

"summary": {

"filepaths": [

"FAIL": 1,

"PASS": 1,

"WAIVED": 0

"severities": {

"High": 1,

"Low": 0,

"Medium": 0,

"Unknown": 0

Fix the Terraform

"Critical": 0,

"Informational": 0,

You can also see the rest of the metadata we defined in the rule earlier.

If you'd like to bring the Terraform into compliance, you can edit ami.tf to replace ami-totallylegitamiid with

ami-09e67e426f25ce0d7 and then run Regula again. We'll just use the default text format this time:

regula run ami.tf --include approved_ami.rego --user-only

"rule_results": {

"ami.tf"

"column": 1

"path": "ami.tf",

"provider": "aws",

"rule_name": "approved_ami",

"resource_id": "aws_instance.bad",

"resource_type": "aws_instance",

"filepath": "ami.tf",

"input_type": "tf",

"provider": "aws",

input.ami represents the ami attribute of each EC2 instance in the input document (your Terraform HCL file - or specifically, a JSON representation of it that Regula transforms behind the scenes). not approved_amis[input.ami] will evaluate to true if the current instance's ami attribute is not a member of

Put together, the deny rule says "If input.ami is not in approved_amis, the Regula check should fail and return an

msg = sprintf("%s is not an approved AMI ID", [input.ami]) This uses the built-in Rego function **sprintf** to print out the **ami** of the currently evaluated resource.

And that's our custom rule! Here it is in full. Now we're going to take Regula for a spin!

Running the custom rule with Regula

not-at-all-suspicious AMIID ami-totallylegitamiid. Run Regula

This command runs our custom rule against the Terraform file, and the --user-only flag says to only apply the

custom rule; for the purposes of this blog post, we're excluding Regula's library of built-in rules. (It's a good practice

CUSTOM_0002: AWS EC2 instances must use approved AMIs [High]

ami-totallylegitamiid is not an approved AMI ID. This is great, because it means our custom rule worked!

Regula showed us that the aws_instance.bad instance (line 13, column 1) did not have an approved AMI ID.

regula run ami.tf --include approved_ami.rego --user-only --format json

"rule_description": "Per company policy, EC2 instances may only use AMI IDs f

"rule_description": "Per company policy, EC2 instances may only use AMI IDs f

"rule_message": "ami-totallylegitamiid is not an approved AMI ID",

"rule_summary": "AWS EC2 instances must use approved AMIs",

In your terminal, cd into the directory containing the code files and run the following command:

to use the built-in rules, though, which are included by default when you execute regula run.)

regula run ami.tf --include approved_ami.rego --user-only

Now, let's test the rule out on a simple Terraform file. The <u>Terraform HCL in our GitHub gist</u> defines two EC2

instances: one "good" instance with the approved AMI ID ami-09e67e426f25ce0d7 and one "bad" instance with the

in ami.tf:13:1 ami-totallylegitamiid is not an approved AMI ID Found one problem.

As you can see, our Terraform file failed the rule with the message

For more details, let's look at the full report, which is formatted as JSON:

Here we can see the full output, including *all* rule results. Below, note how aws_instance.bad has a FAIL result again, whereas aws_instance.good has a PASS rule result:

"rule_id": "CUSTOM_0002", "rule_message": "", "rule_name": "approved_ami", "rule_result": "PASS", "rule_severity": "High", "rule_summary": "AWS EC2 instances must use approved AMIs",

"resource_id": "aws_instance.good",

"resource_type": "aws_instance",

Success! We've secured our Terraform IaC by using only approved AMIs, as proven by Regula. **Further reading**

the Terraform into compliance.

Example: Writing a Simple Rule

IaC security designed for devs

every team can develop, deploy, and operate safely.

Snyk secures your infrastructure as code from SDLC to

runtime in the cloud with a unified policy as code engine so

Writing Rules

<u>Usage</u>

And we see the following output:

No problems found.

Book a live demo

Snyk is a developer security

dependencies, containers, and

industry-leading application and

security intelligence, Snyk puts

Start free

Book a live demo

toolkit.

Posted in: Cloud Security, IaC Security

Product Resources What is Snyk? **Documentation Snyk Code (SAST) Snyk API Docs** platform. Integrating directly into **Snyk Open Source (SCA)** development tools, workflows, and automation pipelines, Snyk makes it **Snyk Container** easy for teams to find, prioritize, and **Snyk Infrastructure as** fix security vulnerabilities in code, Code Blog **Snyk AppRisk (ASPM)** infrastructure as code. Supported by **Developer Security Platform** leaders security expertise in any developer's **Application security** Software supply chain hackers

security

Pricing

DeepCode Al

Integrations

IDE plugins

Deployment options

Secure Al-generated code

API status Careers Disclosed vulnerabilities Events Support portal & FAQ's Press kit **Security fundamentals Resources for security Privacy Resources for ethical Vulnerability Database Snyk OSS Advisor Snyk Top 10 Videos Customer resources**

Customers Contact us Support Report a new vuln **Snyk for government Security & trust** Legal terms For California residents: Do not sell my personal information **Website Terms of Use**

Connect

Book a live demo

Snyk With GitHub Snyk vs Veracode Snyk vs Checkmarx

Security

Application Security

Container Security

Supply Chain Security

JavaScript Security

Open Source Security

AWS Security

Secure SDLC

Security posture

Secure coding

Ethical Hacking

Code Checker

Python

JavaScript

Al in cybersecurity

Enterprise Cybersecurity

8 Expert Tips to **Secure Your Pipelines** Find security issues in the pipeline before you push to production with

Book a live demo

these 8 actionable scanning and integration tips. See the cheatsheet

⊕ EN

Log in

Sign up

In this blog post, we wrote a custom rule in Rego, used Regula to check it against a Terraform HCL file, and brought

If you'd like to learn more about Regula, visit the **Regula docs site**. You might find the following resources useful:

Company

About

© 2024 Snyk Limited Registered in England and Wales

Git Security CI/CD pipelines security **Snyk CLI Snyk Learn Snyk for JavaScript**