Rego Basics

Get hands-on experience with the Language and how it works.

When it comes to cloud security risk assessment, an often overlooked consideration is how your rules are created and what goes into customizing them. Cloud Configuration Rules in Wiz are powered by Open Policy Agent (OPA, pronounced “oh-pah”). OPA provides a high-level declarative language called Rego (pronounced “ray-go”) that lets you define security policies as code. Rego is purpose-built for expressing policies over complex hierarchical data structures. To create custom rules and policies in Wiz, you write them in Rego. Outside of Wiz, Rego is a useful language to learn that can find a place in the course of your work.

Table of Contents

Hello, World!

Policies

In Rego, you define *policies*. Each policy consists of one or more *rules*, which are in turn composed of *assignments* and *conditions*.

Rules, assignments, and conditions define the logic of the policy:

In standard Rego, the output of a policy is either `true` or `false` (booleans).

Hello World—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

From Policies to Rules

Each rule consists of an assignment, followed by a one or more conditions. The relation between an assignment and its condition(s) is implicitly IF; the assignment is evaluated only if the conditions are `true`.

When defining a rule, you should generally write the assignment first, and then the condition(s). The condition(s) are written between curly braces `{...}`:

```
assignment {
   conditions
}
```

The assignment is the name of a document (essentially, a variable; see below) whose value will be changed if all of the conditions evaluate to `true`. Documents can be named practically anything—”foo”, “allow”, “check_tls_version”, etc.—but it is a good practice to give them meaningful names.

Hello World—Exercise 2

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Rule Structure

This policy consists of a package declaration, and a single rule whose assignment is `allow = true`, followed by the condition `{1 == 1}`:

```
package play

allow = true {
   1 == 1
}
```

Because one does, indeed, equal one, this rule will always set the value of allow to `true` .

It is also possible to write a rule without conditions. In this case, the condition is implicitly set to `true` and the assignment is always evaluated. For example, the following two rules are equivalent:

```
n = 1
n = 1 {true}
```

Similarly, an assignment can be written without `=`, in which case the value of `true` will be assigned. For example, the following two rules are equivalent:

```
allow {1 == 1}
allow = true {1 == 1}
```

Hello World—Exercise 3

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

More About Rules

A policy can consist of multiple rules, and each rule can include multiple conditions:

```
# This rule sets the value of ‘understanding_rego’ to
# true if all of the conditions are true
# The conditions check if one equals one, and if two
# equals two

understanding_rego = true {
   1 == 1
   2 == 2
}
```

There is an implicit AND between all conditions within the scope of a rule. Thus, if even a single condition evaluates to `false`, the assignment will not occur. On the other hand, there is animplicit OR between multiple rules that could assign a new value to the same document. Thus, ifthe first rule does not make an assignment, the second (or third, and so on) could.

Hello World—Exercise 4

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Variables & Documents

A *document* in Rego functions much like a variable in other languages. The document is the basic method forholding data, handling calculations, and making decisions between data objects. To make a document storedata, you need first to define it. In Rego, there are three ways to define documents.

Constant Definition

The most common way to define a document is with `:=` , which prevents the document from changing for therest of the policy, much like `const` in C++.

```
# Define a scalar value for the document a
a := 1

# Define a string value for the document b
b := “Rego”

# Define collection of values (set, arrays, objects
# for the document c
c := {“company_name”:”Wiz”, “office”:[“tlv”,”nyc”]}
```

Variables & Documents—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

Default Definition

When multiple rules share the same documents, the shared documents should be defined using the `default` method. This method is valid only in the scope of the assignment, and the document can be changed later in the policy.

```
# Define a default scalar value for the document d
default d = 1

# Define a boolean value for the document allow
default allow = true

# Define the null value for the document e
default e = null
```

Variables & Documents—Exercise 2

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

Equal Definition

The method `=` defines a document and has the same behavior as the method `:=` except:

- In the scope of an assignment, this method can override documents defined with a default method:

```
# Override the default boolean value of “allow”:
default allow = false
allow = true {...}
```

Variables & Documents—Exercise 3

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

- In the scope of a condition, if the document is already defined, `=` behaves the same as the comparison operator `==` and returns `true`.

```
# Define the value of 1 for the document g
g = 1

# Check in the condition if the document g is equal to
# 1
# The document g is equal to 1, and true is returned to
# the assignment
allow {g = 1}
```

Variables & Documents—Exercise 4

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

Definition Requirements

- Each document can be defined only once per rule scope:

```
# Wrong
g := 1
g := 2
# 1 error occurred: rego_type_error: rule named g
# redeclared at policy

# Correct
default g = 1
g = 2
```

- Avoid declaring documents with the same names in different scopes. It might not generatean error, but it is hard to follow:

```
# Confusing
default allow = false
a := 1
allow = true {
   a := 2
   a == 1
}
```

At first, a is 1 and allow is false. Then, in the scope of the rule *another* a is defined to be 2, sothe next line evaluates to false and the overall rule does not change the value of allow. Confusing!

- Use `=` only for overriding document values that were defined using the `default` method:

```
# Wrong:
default allow = false
a := 1
allow = true {
   a = 1
}

# Correct:
default allow = false
a := 1
allow = true {
a == 1
}
```

Variables & Documents—Exercise 5

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Basic Syntax

Rego supports all common data and collection types found in other languages.

Scalar Values

Scalar values are the simplest types in Rego. Scalar values can be strings, numbers, booleans, or null:

- Strings—Characters surrounded by double quotes. In such strings, certain characters must be escaped to appear in the string, such as double quotes themselves, backslashes, etc.

```
s := “s is string”

t := “t is a string that has \”double quotes\” in it”
```

Basic Syntax—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

- Numbers—There is no distinction between a decimal number (float) and an integer (int),they are all numbers.

```
x := 3.14
y := 2
z := 0.5
```

Basic Syntax—Exercise 2

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

- Boolean—Either `true` and `false`. Also, true and false are reserved words that cannot be changed.

```
allow := true
deny := false
```

Basic Syntax—Exercise 3

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

- Null—A built-in constant that has a value of nothing.

```
location := null      # location is nowhere
```

Basic Syntax—Exercise 4

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

Composite Values

Composite values define collections of values, including other composite values, scalar values,or both. All composite values can be indexed starting from zero.

- In simple cases, composite values can be treated as constants like scalar values:

```
user_details_dict := {“user”: “John”, “company”: “Wiz”}
user_details_list := [“John”, “Wiz”]
```

Basic Syntax—Exercise 5

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Basic Syntax—Exercise 6

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

Rego differentiates between **sets** and **arrays**. The two types have some features in common, but there are a few important distinctions:

	Sets	Arrays
Stored Values	Must all be the same type	Must all be the same type
Add/Remove Elements	☑	☑
Ordered	✗	☑
Duplicate Elements	✗	☑
Defined By	Curly Braces { }	Square Brackets [ ]
Keyed	✗	☑
Indexable	✗	☑

- **Sets**—Unordered collections of unique values:

```
x := {1,2,3}
y := {3,2,1}
z := {“a”, “b”, “c”}

x[3]            # evaluates to true because the number
# 3 is a member of x
y[0]            # evaluates to false because the number
# 0 is not a member of y
x == y          # evaluates to true because x and y
# contain the same numbers; order doesn’t matter
x != z          # evaluates to true because x and z do
# not contain the same members
x == z          # error! you can’t compare sets with
# different types of values
```

Basic Syntax—Exercise 7

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

- **Arrays**—Ordered collections of values:

```
x := [1,1,3]
y := [3,1,1]
z := [“a”, “b”, “a”]

x[3]            # evaluates to undefined because there
# is no 3 index in the x (index starts from 0
y[0]            # evaluates to true and returns 3
# because the index 0 is in y
x == y          # evaluates to false because x and y
# contain the same numbers but order different
x != z          # evaluates to true because x and z do
# not contain the same members
x == z          # error! you can’t compare arrays with
# different types of values
```

Basic Syntax—Exercise 8

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Dictionaries

A dictionary is a key-value hash table, aka “dict”. The contents of a dict can be written as aseries of key:value pairs within braces { }:

```
dict = {key1:value1, key2:value2, ... }
```

Whenever possible, use dictionaries instead of nested arrays:

```
# DO NOT COPY THIS. This is an example of what NOT to
# do.
# Array of objects where each object has a unique
# identifier
d := [{“id”: “a123”, “first”: “alice”, “last”:
“smith”},
      {“id”: “a456”, “first”: “bob”, “last”: “jones”},
      {“id”: “a789”, “first”: “clarice”, “last”:
“johnson”}
      ]
# Search through all elements of the array to find the
# ID
d[i].id == “a789”
d[i].first ...
```

Basic Syntax—Exercise 9

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

Instead, use a dictionary where the key is the ID and the value is the first-name/last-name. Given the ID, you can look up the name information directly:

```
# DO THIS INSTEAD OF THE ABOVE. This is an example of
# what you SHOULD do.
# Use object whose keys are the IDs for the objects.
# Looking up an object given its ID requires NO search
d := {“a123”: {“first”: “alice”, “last”: “smith”},
      “a456”: {“first”: “bob”, “last”: “jones”},
      “a789”: {“first”: “clarice”, “last”: “johnson”}
      }
# no search required
d[“a789”].first ...
```

Basic Syntax—Exercise 10

Follow the instructions in the comments to change the OUTPUT from `error` to `true`.

Objects

Objects are unordered key-value collections. Any value type can be used as an object key. Forexample, the following assignment maps port numbers to a list of IP addresses (represented asstrings):

```
ips_by_port := {
   80: [“1.1.1.1”, “1.1.1.2”],
   443: [“2.2.2.1”],
}

ips_by_port[80]
```

Basic Syntax—Exercise 11

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Logical Evaluations

Every line of code in a policy performs a logical evaluation that returns either `true` or `false`. If the conditions in a rule never evaluate to true, the result is undefined. As a result, the document that would otherwise be defined by the rule's assignment remains undefined.

```
# Check if the value of foo equals the value bar, and
# returns a logical true
# Since the condition is true (foo and bar both equal
# 1), the assignment is performed
# The assignment overrides the default value of allow,
# changing it to true
default allow = false
foo := 1
bar := 1
allow = true {
   foo == bar
}

# The string “hello” does not equal the string “world”
# Since the document undefinedDocument was not
# previously defined, it remains undefined
undefinedDocument { “hello” == “world” }
```

Logical Evaluation—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Logical Evaluation—Exercise 2

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

Logical Evaluation—Exercise 3

Follow the instructions in the comments to change the OUTPUT from `false` to `true`.

References & External Input

References are used to access nested documents:

```
# Define an employee array and default value for allow
employee := [“foo”, “bar”, “john”]
default allow = false

# Check in the condition if the first object in the
# array equals to “foo”
allow {employee[0] == “foo”}

# The condition is true, so the assignment is
# performed, overriding the default value of all
```

When using Rego to write Cloud Configuration Rules, you essentially always need external inputin the form of a JSON that contains a cloud resource’s properties, an IaC template, etc. When reading external JSON files or other input , you must start your call with `input` . You can refer to data in the input using the `.` (dot) operator.

Assume the external input is:

```
{ “SecurityGroups”:[
   {“GroupId”:“xyz”},
   {“GroupName”:“myGroup”}
 ]
}
```

You can call the value “myGroup” from the JSON input:

```
input.SecurityGroups[1].GroupName
```

The dot `.` operator (aka interpunction) can be used only when a key is alphanumeric starting with a letter.

Dot `.` is shorthand for brackets `[]`. In other words, Rego makes `x.y` into `x["y"]`. For example:

```
# Assume the external input is the same JSON as above

# Call the value "myGroup" of the JSON input with
# brackets instead of dot:
input["SecurityGroups"][1]["GroupName"]
```

JSON Input—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `"fail"` to `"pass"`.

JSON Input—Exercise 2

Follow the instructions in the comments to change the OUTPUT from `"fail"` to `"pass"`.

Operators & Logical Expressions

Rego offers many built-in logical expressions:

```
# These are logical expressions
x == y # x is equal to y ({1,3,1,4} == {4,4,1,3,1}
#true)
x != y # x is not equal to y
x < y # x is less than y
x <= y # x is less than or equal to y

# Built-in set functions
s3 := s1 & s2 # s3 is the intersection of s1 and s2
s3 := s1 | s2 # s3 is the union of s1 and s2
s3 := s1 - s2 # s3 is the elements in s1 that are not
in s2
```

One especially useful operator is `not`, which has several important behaviors:

- `not` turns undefined into `true`
- `not` turns false into `true`
- `not` turns everything else into `true`

```
# Check if path does not exist
not input.foo.bar

# This is a negation of the return value of the called
function
not myfunction
```

When a path is missing, the result is undefined, which is not an error!

Functions—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `"error"` to `"pass"`.

Functions—Exercise 2

Follow the instructions in the comments to change the OUTPUT from `"error"` to `"pass"`.

Functions—Exercise 3

Follow the instructions in the comments to change the OUTPUT from `"fail"` to `"pass"`.

Looping & Iteration

Iterating through arrays and dictionaries is one of Rego's main strengths.

Looping through arrays is as simple as referencing the index of the array with a symbol. Checkfor yourself what output the following policy generates:

```
# Policy
default allow1 = false
default allow2 = false
default allow3 = false
default allow4 = false
employee := [“foo”, “bar”, “john”]

allow1 {
   employee[x] == “foo”
}

allow2 {
   # The underscore iterator is special, it is
explained below
   employee[_] == “bar”
}

allow3 {
   employee[j] == “john”
}

allow4 {
   employee[i] == “timmy”
}

# Expected output
{
    "allow1": true,
    "allow2": true,
    "allow3": true,
    "allow4": false,
    "employee": [
        "foo",
        "bar",
        "john"
    ]
}
```

The iteration symbol can be declared like a document. It must be alphanumeric and begin with a letter. For example: `i`, `j`, `x`, `test`, or `timmy123`.
Also, the iteration symbol is case sensitive, so `j` is different from `J`.

Referencing Iteration Symbols

Another strength of iteration in Rego is the ability to reference the iteration symbol for advanced use cases. Check for yourself what output the following policy generates:

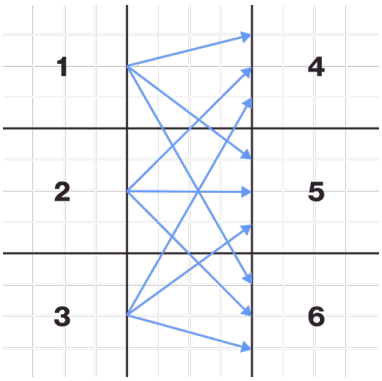
```
# Policy
package play

default allowSameSymbol = false
default allowDifferentSymbols = false

allowSameSymbol {
   input.dogArray[j].firstName == “Timmy”
   input.dogArray[j].lastName == “Reznik”
}

allowDifferentSymbols {
input.dogArray[j].firstName == “Timmy”
input.dogArray[k].lastName == “Reznik”
}

# Input
{
    “dogArray”: [
        {
            “firstName”: “Timmy”,
            “lastName”: “Mymon”
        },
        {
            “firstName”: “Mongo”,
            “lastName”: “Reznik”
        },
        {
            “firstName”: “Babar”,
            “lastName”: “Berkovitz”
        }
    ]
}

# Expected Output
{
    “allowDifferentSymbols”: true,
```

```
    “allowSameSymbol”: false
}
```

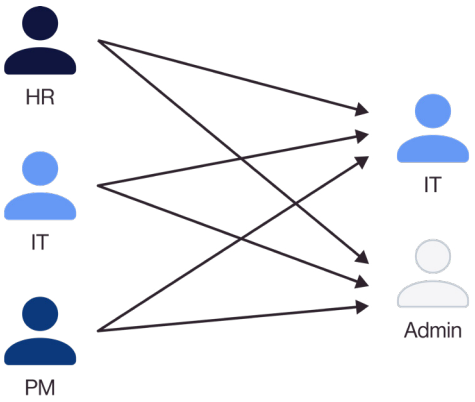
Cross-Array Comparison & Looping

It is sometimes necessary to compare multiple values from many arrays and check everypossible combination. In this case, you must use different symbols for each iteration:

```
# Iterate arr1, and arr2 and compare all combinations
# of values
arr1 = [1,2,3]
arr2 = [4,5,6]
allow {
   arr1[i] == arr2[j]
}
```

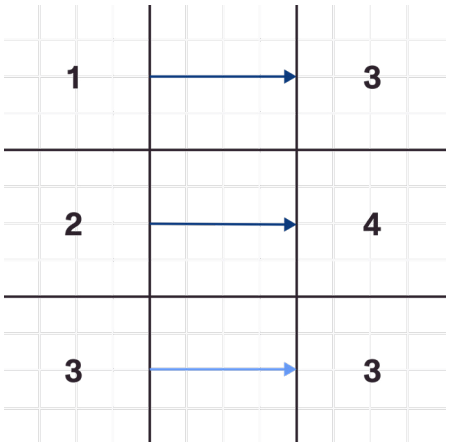



```
# This iterates across the arrays employeeGroup and
# groupWithHighPermissions
# The condition checks if there is a group in the input
# JSON that also exists in the group
# with high permissions document

# input JSON {“employeeGroup”: [“HR”,”IT”,”PM”]}
default rootPermissions = false
groupWithHighPermissions := [“IT”, “Admin”]
rootPermissions = true {
   input[“employeeGroup”][i] ==
groupWithHighPermissions[j]
}
```



It is also sometimes necessary to compare values corresponding to the same index in multiple arrays:

```
# Iterates arr1 and arr2 to find all indices i that are
# equal
# Compare values based on the index:
arr1 = [1,2,3]
arr2 = [3,4,3]
allow {
   arr1[i] == arr2[i]
}
```




```
# Iterates the input JSON employeeGroup and compares to
# the HighPermissions document
# When it finds a match, the condition returns true
# The assignment redefines the value to the document
# “adminUser”
# The value is based on the input JSON employeeName in
# the same index of the match
```

```
# input JSON [{“employeeName”:”bar”, “employeeGroup”:
# “PM”},
#         {“employeeName”:”foo”, “employeeGroup”:
# “IT”}]

default adminUser = null
HighPermissions := [“IT”, “Admin”]
adminUser = input[i][“employeeName”] {
   input[i][“employeeGroup”] = HighPermissions[j]
}
```

If you do not need to use the index of an iterator outside the condition, it is preferrable toreplace it with an underscore `_` symbol. Underscore is the only valid non-letter symbol. It behaves just like any other letter except that unlike the letters it does not keep the indexposition:

```
# check if “foo” exists in the employee array
default allow = false
employee := [“foo”, “bar”, “john”]
allow {employee[_] == “foo”}

# Compare all combinations of values
arr1 = [1,2,3]
arr2 = [3,4,3]
allow {
   arr1[_] == arr2[_]
}
```

In general, you should choose symbols that indicate the purpose of the iteration.

```
# The iteration symbol “name” indicates that the
# iteration searches for equal names
default allow = false
employee := [“foo”, “bar”, “john”]
allow {
   employee[name] == “foo”
}
```

Rego supplies concise and powerful iteration methods, but they can be a bit confusing at first:

```
# check if the user is an admin and a member in the
# admins group

default result = “fail”
user := “john”
user_group := [ {“john”:[“product”,”allUsers”]},
                {“dan”:[“root”,”allUsers”]},
                {“amit”:[“allUsers”]}]
admins := [“john”, “dan”, “noam”]
admins_group := [“admins”, “root”]

result = “pass” {
   admins[i] == user
   admins_group[j] == user_group[_][user][_]
}

# For the user john, the result is “fail”
# For the user dan, the result is “pass”
```

Looping—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `"error"` to `"pass"`.

Looping—Exercise 2

Follow the instructions in the comments to change the OUTPUT from `"error"` to `"pass"`.

Looping—Exercise 3

Follow the instructions in the comments to change the OUTPUT from `"error"` to `"pass"`.

Referencing Rules

You can assign a name to a Rego rule, and then reference it later in your code. This is similar towriting functions in other programing languages, and allows you to create modular and reusablecode pieces.

```
default result = “pass”
default retentionDaysLessThan90 = false

retentionDaysLessThan90 = true {
   input.ServerBlobAuditingPolicy.properties.
retentionDays < 90
}

result = “fail” {
    retentionDaysLessThan90
}
```

The first rule does not change the value of the result document, it only checks a property. Thesecond rule calls the first, which returns the boolean value of the retention `DaysLessThan90` document.

Referencing Rules—Exercise 1

Follow the instructions in the comments to change the OUTPUT from `"error"` to `"pass"`.

Built-In Functions

Rego provides a wide variety of built-in functions to make it easier to perform commonoperations like manipulating scalar values and aggregation. This is only a small selection of theavailable built-in functions.

Time

Basic functions relating to time:

```
time.now_ns() # current time since epoch in nanoseconds
time.parse_duration_ns(duration) # duration valid time
# units:"s","m","h"("-1h","2h4m")
time.parse_rfc3339_ns(value) # representing the time
# value in nanoseconds since epoch
```

For instance, if you wanted to check that secret keys are rotated at least every 90 days:

```
default result = "pass"

now_ns : = time.now_ns()
ninty_days_ns := time.parse_duration_ns("2160h") # 90d
in hours

result = "fail"{
   (now_ns - time.parse_rfc3339_ns(input.
timeCreated))>ninty_days_ns
}
```

Net

Basic functions for manipulating IP addresses:

```
net.cidr_contains(cidr, cidr_or_ip) # true if 'cidr_or_
# ip' is contained within 'cidr'
net.cidr_intersects(cidr1, cidr2) # true if 'cidr1'
# overlaps with 'cidr2'
net.cidr_merge(cidrs_or_ips) # merging the provided
```

```
# list of IP addresses
net.cidr_merge([“192.0.128.0/24”, “192.0.129.0/24”])
# generates {“192.0.128.0/23”}
```

Unmarshal

Convert a JSON in string format to standard JSON so that keys and values can be queried:

```
jsonStr := "{\"Id\":\"\",\"Statement\":[{\"Action\":\"*\",\"Ef
fect\":\"Allow\",\"Principal\":{\"AWS\":\"*\"},\"Resource\":
\"*\"}],\"Version\":\"2012-10-17\"}"
json.unmarshal(jsonStr) # deserialized JSON string to JSON
# Return the following JSON:
{
  "Id": "",
  "Statement": [
    {
      "Action": "*",
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Resource": "*"
    }
  ],
  "Version": "2012-10-17"
}
```

Reserved Names

The following words are reserved and cannot be used for document names:

```
as
default
else
false
import
package
not
null
some
true
with
```

Additional Resources

- Online courses from Styra Academy
- A Rego style guide, also by Styra Academy
- Official Rego documentation from OPA