User Guide

AWS CloudFormation Guard

AWS CloudFormation Guard: User Guide

Table of Contents

What is AWS CloudFormation Guard? **1**
- Are you a first-time Guard user? 1
- Guard features 2
- Using Guard with CloudFormation Hooks 2
- Accessing Guard 3
- Best practices 3

Setting up Guard **4**
- For Linux and macOS 4
  - Install Guard from a pre-built release binary 4
  - Install Guard from Cargo 5
  - Install Guard from Homebrew 6
- For Windows 6
  - Prerequisites 6
  - Install Guard from Cargo 5
  - Install Guard from Chocolatey 8
- As an AWS Lambda function 8
  - Prerequisites 8
  - Install the Rust package manager 9
  - To install Guard as a Lambda function 9
  - To build and run 10
  - Calling the Lambda function 11

Prerequisites and overview for using Guard rules **12**
- Prerequisites 12
- Overview of using Guard rules 12
- Writing Guard rules 13
  - Clauses 13
  - Using queries in clauses 15
  - Using operators in clauses 16
  - Using custom messages in clauses 19
  - Combining clauses 20
  - Using blocks with Guard rules 21
  - Using built-in functions 24
  - Defining queries and filtering 25
  - Assigning and referencing variables in Guard rules 39

Composing named-rule blocks 46
Writing clauses to perform context-aware evaluations 52
Testing Guard rules 65
Prerequisites 65
Overview 65
Walkthrough 67
Using input parameters with Guard rules 76
How to use 76
Example usage 76
Multiple input parameters 78
Validating input data against Guard rules 78
Prerequisites 78
Using the `validate` command 78
Validating multiple rules against multiple data files 79
Troubleshooting Guard 81
Clause fails when no resources of the selected type are present 81
Guard does not evaluate CloudFormation template 81
General troubleshooting topics 82
Guard CLI reference 83
Guard CLI global parameters 83
parse-tree 83
Syntax 83
Parameters 84
Options 84
Examples 84
rulegen 84
Syntax 84
Parameters 85
Options 85
Examples 85
test 85
Syntax 85
Parameters 86
Options 86
Examples 87
Output 87

- See also ... 87
- validate ... 87
  - Syntax ... 87
  - Parameters ... 87
  - Options ... 89
  - Example ... 90
  - Output ... 90
  - See also ... 91

Security ... 92

Document history ... 93

AWS Glossary ... 96

What is AWS CloudFormation Guard?

AWS CloudFormation Guard is an open-source, general-purpose, policy-as-code evaluation tool. The Guard command line interface (CLI) provides a simple-to-use and declarative domain-specific language (DSL) that you can use to express policy as code. In addition, you can use CLI commands to validate structured hierarchical JSON or YAML data against those rules. Guard also provides a built-in unit testing framework to verify that your rules work as intended.

Guard doesn't validate CloudFormation templates for valid syntax or allowed property values. You can use the cfn-lint tool to perform a thorough inspection of template structure.

Guard doesn’t provide server-side enforcement. You can use the CloudFormation Hooks to perform server-side validation and enforcement, where you can block or warn an operation.

For detailed information about AWS CloudFormation Guard development, refer to the Guard GitHub repository.

Topics

- Are you a first-time Guard user?
- Guard features
- Using Guard with CloudFormation Hooks
- Accessing Guard
- Best practices

Are you a first-time Guard user?

If you're a first-time user of Guard, we recommend that you begin by reading the following sections:

- Setting up Guard – This section describes how to install Guard. With Guard, you can write policy rules using the Guard DSL and validate your JSON- or YAML-formatted structured data against those rules.
- Writing Guard rules – This section provides detailed walkthroughs for writing policy rules.
- Testing Guard rules – This section provides a detailed walkthrough for testing your rules to verify that they work as intended, and validating your JSON- or YAML-formatted structured data against your rules.

- Validating input data against Guard rules – This section provides a detailed walkthrough for validating your JSON- or YAML-formatted structured data against your rules.
- Guard CLI reference – This section describes the commands that are available in the Guard CLI.

Guard features

Using Guard, you can write policy rules to validate any JSON- or YAML-formatted structured data against, including but not limited to CloudFormation templates. Guard supports the entire spectrum of end-to-end evaluation of policy checks. Rules are useful in the following business domains:

- **Preventative governance and compliance (shift-left testing)** – Validate infrastructure as code (IaC) or infrastructure and service compositions against policy rules that represent your organizational best practices for security and compliance. For example, you can validate CloudFormation templates, CloudFormation change sets, JSON-based Terraform configuration files, or Kubernetes configurations.
- **Detective governance and compliance** – Validate conformity of Configuration Management Database (CMDB) resources such as AWS Config-based configuration items (CIs). For example, developers can use Guard policies against AWS Config CIs to continuously monitor the state of deployed AWS and non-AWS resources, detect violations from policies, and start remediation.
- **Deployment safety** – Ensure that changes are safe before deployment. For example, validate CloudFormation change sets against policy rules to prevent changes that result in resource replacement, such as renaming an Amazon DynamoDB table.

Using Guard with CloudFormation Hooks

You can use CloudFormation Guard to author a Hook in CloudFormation Hooks. CloudFormation Hooks allows you to proactively enforce your Guard rules before CloudFormation create, update, or delete operations and AWS Cloud Control API create or update operations. Hooks ensure your resource configurations are compliant with your organization's security, operational, and cost optimization best practices.

For details on how to use Guard to author CloudFormation Guard Hooks, see Write Guard rules to evaluate resources for Guard Hooks in the *CloudFormation Hooks User Guide*.

Accessing Guard

To access the Guard DSL and commands, you must install the Guard CLI. For information about installing the Guard CLI, see Setting up Guard.

Best practices

Write simple rules, and use named rules to reference them in other rules. Complex rules can be difficult to maintain and test.

Setting up AWS CloudFormation Guard

AWS CloudFormation Guard is an open-source command line interface (CLI). It provides you with a simple, domain-specific language to write policy rules and validate their structured hierarchical JSON and YAML data against those rules. The rules can represent company policy guidelines related to security, compliance, and more. The structured hierarchical data can represent cloud infrastructure described as code. For example, you can create rules to ensure that they always model encrypted Amazon Simple Storage Service (Amazon S3) buckets in their CloudFormation templates.

The following topics provide information about how to install Guard using your chosen operating system or as an AWS Lambda function.

Topics

- Installing Guard for Linux and macOS
- Installing Guard for Windows
- Installing Guard as an AWS Lambda function

Installing Guard for Linux and macOS

You can install AWS CloudFormation Guard for Linux and macOS by using the pre-built release binary, Cargo, or through Homebrew.

Install Guard from a pre-built release binary

Use the following procedure to install Guard from a pre-built binary.

1. Open a terminal, and run the following command.

```
curl --proto '=https' --tlsv1.2 -sSf https://raw.githubusercontent.com/aws-
cloudformation/cloudformation-guard/main/install-guard.sh | sh
```

2. Run the following command to set your PATH variable.

```
export PATH=~/.guard/bin:$PATH
```

Results: You have successfully installed Guard and set the PATH variable.

- (Optional) To confirm the installation of Guard, run the following command.

  ```
  cfn-guard --version
  ```

  The command returns the following output.

  ```
  cfn-guard 3.1.2
  ```

Install Guard from Cargo

Cargo is the Rust package manager. Complete the following steps to install Rust, which includes Cargo. Then, install Guard from Cargo.

1. Run the following command from a terminal, and follow the onscreen instructions to install Rust.

   ```
   curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
   ```

   - (Optional) For Ubuntu environments, run the following command.

     ```
     sudo apt-get update; sudo apt install build-essential
     ```

2. Configure your PATH environment variable, and run the following command.

   ```
   source $HOME/.cargo/env
   ```

3. With Cargo installed, run the following command to install Guard.

   ```
   cargo install cfn-guard
   ```

   Results: You have successfully installed Guard.

   - (Optional) To confirm the installation of Guard, run the following command.

     ```
     cfn-guard --version
     ```

     The command returns the following output.

```
cfn-guard 3.1.2
```

Install Guard from Homebrew

Homebrew is a package manager for macOS and Linux. Complete the following steps to install Homebrew. Then, install Guard from Homebrew.

1. Run the following command from a terminal, and follow the onscreen instructions to install Homebrew.

   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
   ```

2. With Homebrew installed, run the following command to install Guard.

   ```
   brew install cloudformation-guard
   ```

   Results: You have successfully installed Guard.

   - (Optional) To confirm the installation of Guard, run the following command.

     ```
     cfn-guard --version
     ```

     The command returns the following output.

     ```
     cfn-guard 3.1.2
     ```

Installing Guard for Windows

You can install AWS CloudFormation Guard for Windows through Cargo or through Chocolatey.

Prerequisites

To build Guard from the command line interface, you must install the Build Tools for Visual Studio 2019.

1. Download Microsoft Visual C++ build tools from the Build Tools for Visual Studio 2019 website.
2. Run the installer, and select the defaults.

Install Guard from Cargo

Cargo is the Rust package manager. Complete the following steps to install Rust, which includes Cargo. Then, install Guard from Cargo.

1. Download Rust and then run **rustup-init.exe**.
2. From the command prompt, choose **1**, which is the default option.

   The command returns the following output.

   ```
   Rust is installed now. Great!

       To get started you may need to restart your current shell.
       This would reload its PATH environment variable to include
       Cargo's bin directory (%USERPROFILE%\.cargo\bin).

       Press the Enter key to continue.
   ```
3. To finish the installation, press the **Enter** key.
4. With Cargo installed, run the following command to install Guard.

   ```
   cargo install cfn-guard
   ```

   Results: You have successfully installed Guard.

   - (Optional) To confirm the installation of Guard, run the following command.

     ```
     cfn-guard --version
     ```

     The command returns the following output.

     ```
     cfn-guard 3.1.2
     ```

Install Guard from Chocolatey

Chocolatey is a package manager for Windows. Complete the following steps to install Chocolatey. Then, install Guard from Chocolatey.

1. Follow this guide to install Chocolatey
2. With Chocolatey installed, run the following command to install Guard.

```
choco install cloudformation-guard
```

Results: You have successfully installed Guard.

- (Optional) To confirm the installation of Guard, run the following command.

```
cfn-guard --version
```

The command returns the following output.

```
cfn-guard 3.1.2
```

Installing Guard as an AWS Lambda function

You can install AWS CloudFormation Guard through Cargo, the Rust package manager. *Guard as an AWS Lambda* function (`cfn-guard-lambda`) is a lightweight wrapper around Guard (`cfn-guard`) that can be used as a Lambda function.

Prerequisites

Before you can install Guard as a Lambda function, you must fulfill the following prerequisites:

- AWS Command Line Interface (AWS CLI) configured with permissions to deploy and invoke Lambda functions. For more information, see Configuring the AWS CLI.
- An AWS Lambda execution role in AWS Identity and Access Management (IAM). For more information, see AWS Lambda execution role.
- In CentOS/RHEL environments, add the `musl-libc` package repository to your yum config. For more information, see ngompa/musl-libc.

Install the Rust package manager

Cargo is the Rust package manager. Complete the following steps to install Rust, which includes Cargo.

1. Run the following command from a terminal, and then follow the onscreen instructions to install Rust.

   ```
   curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
   ```

   - (Optional) For Ubuntu environments, run the following command.

     ```
     sudo apt-get update; sudo apt install build-essential
     ```

2. Configure your PATH environment variable, and run the following command.

   ```
   source $HOME/.cargo/env
   ```

Install Guard as a Lambda function (Linux, macOS, or Unix)

To install Guard as a Lambda function, complete the following steps.

1. From your command terminal, run the following command.

   ```
   cargo install cfn-guard-lambda
   ```

   - (Optional) To confirm the installation of Guard as a Lambda function, run the following command.

     ```
     cfn-guard-lambda --version
     ```

     The command returns the following output.

     ```
     cfn-guard-lambda 3.1.2
     ```

2. To install `musl` support, run the following command.

   ```
   rustup target add x86_64-unknown-linux-musl
   ```

3. Build with `musl`, and then run the following command in your terminal.

```
cargo build --release --target x86_64-unknown-linux-musl
```

   For a custom runtime, AWS Lambda requires an executable with the name `bootstrap` in the deployment package .zip file. Rename the generated `cfn-lambda` executable to `bootstrap` and then add it to the .zip archive.

   - For macOS environments, create your cargo configuration file in the root of the Rust project or in `~/.cargo/config`.

```
[target.x86_64-unknown-linux-musl]
linker = "x86_64-linux-musl-gcc"
```

4. Change to the `cfn-guard-lambda` root directory.

```
cd ~/.cargo/bin/cfn-guard-lambda
```

5. Run the following command in your terminal.

```
cp ./../target/x86_64-unknown-linux-musl/release/cfn-guard-lambda ./bootstrap &&
 zip lambda.zip bootstrap && rm bootstrap
```

6. Run the following command to submit `cfn-guard`as a Lambda function to your account.

```
aws lambda create-function --function-name cfnGuard \
 --handler guard.handler \
 --zip-file fileb://./lambda.zip \
 --runtime provided \
 --role arn:aws:iam::444455556666:role/your_lambda_execution_role \
 --environment Variables={RUST_BACKTRACE=1} \
 --tracing-config Mode=Active
```

To build and run Guard as a Lambda function

To invoke the submitted `cfn-guard-lambda` as a Lambda function, run the following command.

```
aws lambda invoke --function-name cfnGuard \
  --payload '{"data":"input data","rules":["rule1","rule2"]}' \
```

```
output.json
```

To call the Lambda function request structure

Requests to `cfn-guard-lambda` require the following fields:

- `data` – The string version of the YAML or JSON template
- `rules` – The string version of the rule set file

Prerequisites and overview for using Guard rules

This section demonstrates how you can complete the core Guard tasks of writing, testing, and validating rules against JSON- or YAML-formatted data. In addition, it contains detailed walkthroughs that demonstrate writing rules that respond to specific use cases.

Topics

- Prerequisites
- Overview of using Guard rules
- Writing AWS CloudFormation Guard rules
- Testing AWS CloudFormation Guard rules
- Using input parameters with AWS CloudFormation Guard rules
- Validating input data against AWS CloudFormation Guard rules

Prerequisites

Before you can write policy rules using the Guard domain-specific language (DSL), you must install the Guard command line interface (CLI). For more information, see Setting up Guard.

Overview of using Guard rules

When using Guard, you typically perform the following steps:

1. Write JSON- or YAML-formatted data to validate.
2. Write Guard policy rules. For more information, see Writing Guard rules.
3. Verify that your rules work as intended by using the Guard `test` command. For more information about unit testing, see Testing Guard rules.
4. Use the Guard `validate` command to validate your JSON- or YAML-formatted data against your rules. For more information, see Validating input data against Guard rules.

Writing AWS CloudFormation Guard rules

In AWS CloudFormation Guard, *rules* are policy-as-code rules. You write rules in the Guard domain-specific language (DSL) that you can validate your JSON- or YAML-formatted data against. Rules are made up of *clauses*.

You can save rules written using the Guard DSL into plaintext files that use any file extension.

You can create multiple rule files and categorize them as a *rule set*. Rule sets allow you to validate your JSON- or YAML-formatted data against multiple rule files at the same time.

Topics

- Clauses
- Using queries in clauses
- Using operators in clauses
- Using custom messages in clauses
- Combining clauses
- Using blocks with Guard rules
- Using built-in functions
- Defining Guard queries and filtering
- Assigning and referencing variables in Guard rules
- Composing named-rule blocks in AWS CloudFormation Guard
- Writing clauses to perform context-aware evaluations

Clauses

Clauses are Boolean expressions that evaluate to either true (`PASS`) or false (`FAIL`). Clauses use either binary operators to compare two values or unary operators that operate on a single value.

Examples of unary clauses

The following unary clause evaluates whether the collection `TcpBlockedPorts` is empty.

```
InputParameters.TcpBlockedPorts not empty
```

The following unary clause evaluates whether the `ExecutionRoleArn` property is a string.

```
Properties.ExecutionRoleArn is_string
```

Examples of binary clauses

The following binary clause evaluates whether the `BucketName` property contains the string `encrypted`, regardless of casing.

```
Properties.BucketName != /(?i)encrypted/
```

The following binary clause evaluates whether the `ReadCapacityUnits` property is less than or equal to 5,000.

```
Properties.ProvisionedThroughput.ReadCapacityUnits <= 5000
```

Syntax for writing Guard rule clauses

```
<query> <operator> [query|value literal] [custom message]
```

Properties of Guard rule clauses

`query`

A dot (`.`) separated expression written to traverse hierarchical data. Query expressions can include filter expressions to target a subset of values. Queries can be assigned to variables so that you can write them once and reference them elsewhere in a rule set, which will allow you to access query results.

For more information about writing queries and filtering, see Defining queries and filtering.

Required: Yes

`operator`

A unary or binary operator that helps check the state of the query. The left-hand side (LHS) of a binary operator must be a query, and the right-hand side (RHS) must be either a query or a value literal.

Supported binary operators: == (Equal) | != (Not equal) | > (Greater than) | >= (Greater than or equal to) | < (Less than) | <= (Less than or equal to) | `IN` (In a list of form [x, y, z]

Supported unary operators: `exists` | `empty` | `is_string` | `is_list` | `is_struct` | `not(!)`

Required: Yes

`query|value literal`

A query or a supported value literal such as `string` or `integer(64)`.

Supported value literals:

- All primitive types: `string`, `integer(64)`, `float(64)`, `bool`, `char`, `regex`
- All specialized range types for expressing `integer(64)`, `float(64)`, or `char` ranges expressed as:
  - `r[<lower_limit>, <upper_limit>]`, which translates to any value `k` that satisfies the following expression: `lower_limit <= k <= upper_limit`
  - `r[<lower_limit>, <upper_limit>)`, which translates to any value `k` that satisfies the following expression: `lower_limit <= k < upper_limit`
  - `r(<lower_limit>, <upper_limit>]`, which translates to any value `k` that satisfies the following expression: `lower_limit < k <= upper_limit`
  - `r(<lower_limit>, <upper_limit>)`, which translates to any value `k` that satisfies the following expression: `lower_limit < k < upper_limit`
- Associative arrays (maps) for nested key-value structure data. For example:

  ```
  { "my-map": { "nested-maps": [ { "key": 10, "value": 20 } ] } }
  ```
- Arrays of primitive types or associative array types

Required: Conditional; required when a binary operator is used.

`custom message`

A string that provides information about the clause. The message is displayed in the verbose outputs of the `validate` and `test` commands and can be useful for understanding or debugging rule evaluation on hierarchical data.

Required: No

Using queries in clauses

For information about writing queries, see Defining queries and filtering and Assigning and referencing variables in Guard rules.

Using operators in clauses

The following are example CloudFormation templates, `Template-1` and `Template-2`. To demonstrate the use of supported operators, the example queries and clauses in this section refer to these example templates.

Template-1

```
Resources:
 S3Bucket:
   Type: AWS::S3::Bucket
   Properties:
     BucketName: MyServiceS3Bucket
     BucketEncryption:
       ServerSideEncryptionConfiguration:
         - ServerSideEncryptionByDefault:
             SSEAlgorithm: 'aws:kms'
             KMSMasterKeyID: 'arn:aws:kms:us-
east-1:123456789:key/056ea50b-1013-3907-8617-c93e474e400'
     Tags:
       - Key: stage
         Value: prod
       - Key: service
         Value: myService
```

Template-2

```
Resources:
 NewVolume:
   Type: AWS::EC2::Volume
   Properties:
     Size: 100
     VolumeType: io1
     Iops: 100
     AvailabilityZone:
       Fn::Select:
         - 0
         - Fn::GetAZs: us-east-1
     Tags:
       - Key: environment
         Value: test
   DeletionPolicy: Snapshot
```

Examples of clauses that use unary operators

- `empty` – Checks if a collection is empty. You can also use it to check if a query has values in a hierarchical data because queries result in a collection. You can't use it to check whether string value queries have an empty string (`""`) defined. For more information, see Defining queries and filtering.

  The following clause checks whether the template has one or more resources defined. It evaluates to `PASS` because a resource with the logical ID `S3Bucket` is defined in `Template-1`.

  ```
  Resources !empty
  ```

  The following clause checks whether one or more tags are defined for the `S3Bucket` resource. It evaluates to `PASS` because `S3Bucket` has two tags defined for the `Tags` property in `Template-1`.

  ```
  Resources.S3Bucket.Properties.Tags !empty
  ```

- `exists` – Checks whether each occurrence of the query has a value and can be used in place of `!= null`.

  The following clause checks whether the `BucketEncryption` property is defined for the `S3Bucket`. It evaluates to `PASS` because `BucketEncryption` is defined for `S3Bucket` in `Template-1`.

  ```
  Resources.S3Bucket.Properties.BucketEncryption exists
  ```

Note

The `empty` and `not exists` checks evaluate to `true` for missing property keys when traversing the input data. For example, if the `Properties` section isn't defined in the template for the `S3Bucket`, the clause `Resources.S3Bucket.Properties.Tag empty` evaluates to `true`. The `exists` and `empty` checks don't display the JSON pointer path inside the document in the error messages. Both of these clauses often have retrieval errors that don't maintain this traversal information.

- `is_string` – Checks whether each occurrence of the query is of `string` type.

  The following clause checks whether a string value is specified for the `BucketName` property of the `S3Bucket` resource. It evaluates to `PASS` because the string value `"MyServiceS3Bucket"` is specified for `BucketName` in `Template-1`.

  ```
  Resources.S3Bucket.Properties.BucketName is_string
  ```

- `is_list` – Checks whether each occurrence of the query is of `list` type.

  The following clause checks whether a list is specified for the `Tags` property of the `S3Bucket` resource. It evaluates to `PASS` because two key-value pairs are specified for `Tags` in `Template-1`.

  ```
  Resources.S3Bucket.Properties.Tags is_list
  ```

- `is_struct` – Checks whether each occurrence of the query is structured data.

  The following clause checks whether structured data is specified for the `BucketEncryption` property of the `S3Bucket` resource. It evaluates to `PASS` because `BucketEncryption` is specified using the `ServerSideEncryptionConfiguration` property type *`(object)`* in `Template-1`.

  ```
  Resources.S3Bucket.Properties.BucketEncryption is_struct
  ```

> **Note**
>
> To check the inverse state, you can use the (`not !`) operator with the `is_string`, `is_list`, and `is_struct` operators .

Examples of clauses that use binary operators

The following clause checks whether the value specified for the `BucketName` property of the `S3Bucket` resource in `Template-1` contains the string `encrypt`, regardless of casing. This evaluates to `PASS` because the specified bucket name `"MyServiceS3Bucket"` does not contain the string `encrypt`.

```
Resources.S3Bucket.Properties.BucketName != /(?i)encrypt/
```

The following clause checks whether the value specified for the `Size` property of the `NewVolume` resource in `Template-2` is within a specific range: 50 <= `Size` <= 200. It evaluates to `PASS` because `100` is specified for `Size`.

```
Resources.NewVolume.Properties.Size IN r[50,200]
```

The following clause checks whether the value specified for the `VolumeType` property of the `NewVolume` resource in `Template-2` is `io1`, `io2`, or `gp3`. It evaluates to `PASS` because `io1` is specified for `NewVolume`.

```
Resources.NewVolume.Properties.NewVolume.VolumeType IN [ 'io1','io2','gp3' ]
```

Note

The example queries in this section demonstrate the use of operators using the resources with logical IDs `S3Bucket` and `NewVolume`. Resource names are often user-defined and can be arbitrarily named in an infrastructure as code (IaC) template. To write a rule that is generic and applies to all `AWS::S3::Bucket` resources defined in the template, the most common form of query used is `Resources.*[ Type == ‘AWS::S3::Bucket’ ]`. For more information, see Defining queries and filtering for details about usage and explore the examples directory in the `cloudformation-guard` GitHub repository.

Using custom messages in clauses

In the following example, clauses for `Template-2` include a custom message.

```
Resources.NewVolume.Properties.Size IN r(50,200)
<<
    EC2Volume size must be between 50 and 200,
    not including 50 and 200
>>
Resources.NewVolume.Properties.VolumeType IN [ 'io1','io2','gp3' ] <<Allowed Volume
 Types are io1, io2, and gp3>>
```

Combining clauses

In Guard, each clause written on a new line is combined implicitly with the next clause by using conjunction (Boolean and logic). See the following example.

```
# clause_A ^ clause_B ^ clause_C
clause_A
clause_B
clause_C
```

You can also use disjunction to combine a clause with the next clause by specifying `or|OR` at the end of the first clause.

```
<query> <operator> [query|value literal] [custom message] [or|OR]
```

In a Guard clause, disjunctions are evaluated first, followed by conjunctions. Guard rules can be defined as a conjunction of disjunction of clauses (an `and|AND` of `or|OR`s) that evaluate to either `true` (`PASS`) or `false` (`FAIL`). This is similar to [Conjunctive normal form](https://en.wikipedia.org/wiki/Conjunctive_normal_form).

The following examples demonstrate the order of evaluations of clauses.

```
# (clause_E v clause_F) ^ clause_G
clause_E OR clause_F
clause_G

# (clause_H v clause_I) ^ (clause_J v clause_K)
clause_H OR
clause_I
clause_J OR
clause_K

# (clause_L v clause_M v clause_N) ^ clause_O
clause_L OR
clause_M OR
clause_N
clause_O
```

All clauses that are based on the example `Template-1` can be combined by using conjunction. See the following example.

```
Resources.S3Bucket.Properties.BucketName is_string
```

```
Resources.S3Bucket.Properties.BucketName != /(?i)encrypt/
Resources.S3Bucket.Properties.BucketEncryption exists
Resources.S3Bucket.Properties.BucketEncryption is_struct
Resources.S3Bucket.Properties.Tags is_list
Resources.S3Bucket.Properties.Tags !empty
```

Using blocks with Guard rules

Blocks are compositions that remove verbosity and repetition from a set of related clauses, conditions, or rules. There are three types of blocks:

- Query blocks
- `when` blocks
- Named-rule blocks

Query blocks

Following are the clauses that are based on the example `Template-1`. Conjunction was used to combine the clauses.

```
Resources.S3Bucket.Properties.BucketName is_string
Resources.S3Bucket.Properties.BucketName != /(?i)encrypt/
Resources.S3Bucket.Properties.BucketEncryption exists
Resources.S3Bucket.Properties.BucketEncryption is_struct
Resources.S3Bucket.Properties.Tags is_list
Resources.S3Bucket.Properties.Tags !empty
```

Parts of the query expression in each clause are repeated. You can improve composability and remove verbosity and repetition from a set of related clauses with the same initial query path by using a query block. The same set of clauses can be written as shown in the following example.

```
Resources.S3Bucket.Properties {
    BucketName is_string
    BucketName != /(?i)encrypt/
    BucketEncryption exists
    BucketEncryption is_struct
    Tags is_list
    Tags !empty
}
```

In a query block, the query preceding the block sets the context for the clauses inside the block.

For more information about using blocks, see Composing named-rule blocks.

when blocks

You can evaluate blocks conditionally by using `when` blocks, which take the following form.

```
  when <condition> {
       Guard_rule_1
       Guard_rule_2
       ...
   }
```

The `when` keyword designates the start of the `when` block. `condition` is a Guard rule. The block is only evaluated if the evaluation of the condition results in `true` (`PASS`).

The following is an example `when` block that is based on `Template-1`.

```
when Resources.S3Bucket.Properties.BucketName is_string {
     Resources.S3Bucket.Properties.BucketName != /(?i)encrypt/
 }
```

The clause within the `when` block is only evaluated if the value specified for `BucketName` is a string. If the value specified for `BucketName` is referenced in the `Parameters` section of the template as shown in the following example, the clause within the `when` block is not evaluated.

```
Parameters:
   S3BucketName:
     Type: String
 Resources:
   S3Bucket:
     Type: AWS::S3::Bucket
     Properties:
       BucketName:
         Ref: S3BucketName
     ...
```

Named-rule blocks

You can assign a name to a set of rules (*rule set*), and then reference these modular validation blocks, called *named-rule blocks*, in other rules. Named-rule blocks take the following form.

```
rule <rule name> [when <condition>] {
    Guard_rule_1
    Guard_rule_2
    ...
    }
```

The `rule` keyword designates the start of the named-rule block.

`rule name` is a human-readable string that uniquely identifies a named-rule block. It's a label for the Guard rule set that it encapsulates. In this use, the term *Guard rule* includes clauses, query blocks, `when` blocks, and named-rule blocks. The rule name can be used to refer to the evaluation result of the rule set that it encapsulates, which makes named-rule blocks reusable. The rule name also provides context about rule failures in the `validate` and `test` command outputs. The rule name is displayed along with the block's evaluation status (`PASS`, `FAIL`, or `SKIP`) in the evaluation output of the rules file. See the following example.

```
# Sample output of an evaluation where check1, check2, and check3 are rule names.
template.json Status = **FAIL**
**SKIP rules**
check1 **SKIP**
**PASS rules**
check2 **PASS**
**FAILED rules**
check3 **FAIL**
```

You can also evaluate named-rule blocks conditionally by specifying the `when` keyword followed by a condition after the rule name.

Following is the example `when` block that was discussed previously in this topic.

```
rule checkBucketNameStringValue when Resources.S3Bucket.Properties.BucketName is_string
 {
    Resources.S3Bucket.Properties.BucketName != /(?i)encrypt/
}
```

Using named-rule blocks, the preceding can also be written as follows.

```
rule checkBucketNameIsString {
    Resources.S3Bucket.Properties.BucketName is_string
```

```
}
rule checkBucketNameStringValue when checkBucketNameIsString {
    Resources.S3Bucket.Properties.BucketName != /(?i)encrypt/
}
```

You can reuse and group named-rule blocks with other Guard rules. Following are a few examples.

```
rule rule_name_A {
    Guard_rule_1 OR
    Guard_rule_2
    ...
}

rule rule_name_B {
    Guard_rule_3
    Guard_rule_4
    ...
}

rule rule_name_C {
    rule_name_A OR rule_name_B
}

rule rule_name_D {
    rule_name_A
    rule_name_B
}

rule rule_name_E when rule_name_D {
    Guard_rule_5
    Guard_rule_6
    ...
}
```

Using built-in functions

AWS CloudFormation Guard provides built-in functions that you can use in your rules to perform operations such as string manipulation, JSON parsing, and data type conversion. Functions are supported only through assignment to a variable.

Key functions

`json_parse(json_string)`

Parses inline JSON strings from a template. After parsing, you can evaluate properties of the resulting object.

`count(collection)`

Returns the number of items that a query resolves to.

`regex_replace(base_string, regex_to_extract, regex_replacement)`

Replaces parts of a string using regular expressions.

For a complete list of available functions including string manipulation, collection operations, and data type conversion functions, see the Functions documentation in the Guard GitHub repository.

Defining Guard queries and filtering

This topic covers writing queries and using filtering when writing Guard rule clauses.

Prerequisites

Filtering is an advanced AWS CloudFormation Guard concept. We recommend that you review the following foundational topics before you learn about filtering:

- What is AWS CloudFormation Guard?
- Writing rules, clauses

Defining queries

Query expressions are simple dot (`.`) separated expressions written to traverse hierarchical data. Query expressions can include filter expressions to target a subset of values. When queries are evaluated, they result in a collection of values, similar to a result set returned from an SQL query.

The following example query searches a CloudFormation template for `AWS::IAM::Role` resources.

```
Resources.*[ Type == 'AWS::IAM::Role' ]
```

Queries follow these basic principles:

- Each dot (`.`) part of the query traverses down the hierarchy when an explicit key term is used, such as `Resources` or `Properties.Encrypted.` If any part of the query doesn't match the incoming datum, Guard throws a retrieval error.
- A dot (`.`) part of the query that uses a wildcard `*` traverses all values for the structure at that level.
- A dot (`.`) part of the query that uses an array wildcard `[*]` traverses all indices for that array.
- All collections can be filtered by specifying filters inside square brackets `[]`. Collections can be encountered in the following ways:
  - Naturally occurring arrays in datum are collections. Following are examples:

    Ports: `[20, 21, 110, 190]`

    Tags: `[{"Key": "Stage", "Value": "PROD"}, {"Key": "App", "Value": "MyService"}]`
  - When traversing all values for a structure like `Resources.*`
  - Any query result is itself a collection from which values can be further filtered. See the following example.

    ```
    # Query all resources
    let all_resources = Resource.*

    # Filter IAM resources from query results
    let iam_resources = %resources[ Type == /IAM/ ]

    # Further refine to get managed policies
    let managed_policies = %iam_resources[ Type == /ManagedPolicy/ ]

    # Traverse each managed policy
    %managed_policies {
        # Do something with each policy
    }
    ```

The following is an example CloudFormation template snippet.

```
Resources:
  SampleRole:
```

```
    Type: AWS::IAM::Role
    ...
  SampleInstance:
    Type: AWS::EC2::Instance
    ...
  SampleVPC:
     Type: AWS::EC2::VPC
    ...
  SampleSubnet1:
    Type: AWS::EC2::Subnet
    ...
  SampleSubnet2:
    Type: AWS::EC2::Subnet
    ...
```

Based on this template, the path traversed is `SampleRole` and the final value selected is `Type: AWS::IAM::Role`.

```
Resources:
  SampleRole:
    Type: AWS::IAM::Role
    ...
```

The resulting value of the query `Resources.*[ Type == 'AWS::IAM::Role' ]` in YAML format is shown in the following example.

```
- Type: AWS::IAM::Role
  ...
```

Some of the ways that you can use queries are as follows:

- Assign a query to variables so that query results can be accessed by referencing those variables.
- Follow the query with a block that tests against each of the selected values.
- Compare a query directly against a basic clause.

Assigning queries to variables

Guard supports one-shot variable assignments within a given scope. For more information about variables in Guard rules, see Assigning and referencing variables in Guard rules.

You can assign queries to variables so that you can write queries once and then reference them elsewhere in your Guard rules. See the following example variable assignments that demonstrate query principles discussed later in this section.

```
#
# Simple query assignment
#
let resources = Resources.* # All resources

#
# A more complex query here (this will be explained below)
#
let iam_policies_allowing_log_creates = Resources.*[
    Type in [/IAM::Policy/, /IAM::ManagedPolicy/]
    some Properties.PolicyDocument.Statement[*] {
         some Action[*] == 'cloudwatch:CreateLogGroup'
         Effect == 'Allow'
    }
]
```

Directly looping through values from a variable assigned to a query

Guard supports directly running against the results from a query. In the following example, the `when` block tests against the `Encrypted`, `VolumeType`, and `AvailabilityZone` property for each `AWS::EC2::Volume` resource found in a CloudFormation template.

```
let ec2_volumes = Resources.*[ Type == 'AWS::EC2::Volume' ]

when %ec2_volumes !empty {
    %ec2_volumes {
        Properties {
            Encrypted == true
            VolumeType in ['gp2', 'gp3']
            AvailabilityZone in ['us-west-2b', 'us-west-2c']
        }
    }
}
```

Direct clause-level comparisons

Guard also supports queries as a part of direct comparisons. For example, see the following.

```
let resources = Resources.*

    some %resources.Properties.Tags[*].Key == /PROD$/
    some %resources.Properties.Tags[*].Value == /^App/
```

In the preceding example, the two clauses (starting with the `some` keyword) expressed in the form shown are considered independent clauses and are evaluated separately.

Single clause and block clause form

Taken together, the two example clauses shown in the preceding section aren't equivalent to the following block.

```
let resources = Resources.*

some %resources.Properties.Tags[*] {
    Key == /PROD$/
    Value == /^App/
}
```

This block queries for each `Tag` value in the collection and compares its property values to the expected property values. The combined form of the clauses in the preceding section evaluates the two clauses independently. Consider the following input.

```
Resources:
  ...
  MyResource:
    ...
    Properties:
      Tags:
        - Key: EndPROD
          Value: NotAppStart
        - Key: NotPRODEnd
          Value: AppStart
```

Clauses in the first form evaluate to `PASS`. When validating the first clause in first form, the following path across `Resources`, `Properties`, `Tags`, and `Key` matches the value `NotPRODEnd` and does not match the expected value `PROD`.

```
Resources:
```

```
  ...
  MyResource:
    ...
    Properties:
      Tags:
        - Key: EndPROD
          Value: NotAppStart
        - Key: NotPRODEnd
          Value: AppStart
```

The same happens with the second clause of the first form. The path across `Resources`, `Properties`, `Tags`, and `Value` matches the value `AppStart`. As a result, the second clause independently.

The overall result is a `PASS`.

However, the block form evaluates as follows. For each `Tags` value, it compares if both the `Key` and `Value` does match; `NotAppStart` and `NotPRODEnd` values are not matched in the following example.

```
Resources:
  ...
  MyResource:
    ...
    Properties:
      Tags:
        - Key: EndPROD
          Value: NotAppStart
        - Key: NotPRODEnd
          Value: AppStart
```

Because evaluations check for both `Key == /PROD$/`, and `Value == /^App/`, the match is not complete. Therefore, the result is `FAIL`.

> **Note**
>
> When working with collections, we recommend that you use the block clause form when you want to compare multiple values for each element in the collection. Use the single clause form when the collection is a set of scalar values, or when you only intend to compare a single attribute.

Query outcomes and associated clauses

All queries return a list of values. Any part of a traversal, such as a missing key, empty values for an array (`Tags: []`) when accessing all indices, or missing values for a map when encountering an empty map (`Resources: {}`), can lead to retrieval errors.

All retrieval errors are considered failures when evaluating clauses against such queries. The only exception is when explicit filters are used in the query. When filters are used, associated clauses are skipped.

The following block failures are associated with running queries.

- If a template does not contain resources, then the query evaluates to `FAIL`, and the associated block level clauses also evaluate to `FAIL`.
- When a template contains an empty resources block like `{ "Resources": {} }`, the query evaluates to `FAIL`, and the associated block level clauses also evaluate to `FAIL`.
- If a template contains resources but none match the query, then the query returns empty results, and the block level clauses are skipped.

Using filters in queries

Filters in queries are effectively Guard clauses that are used as selection criteria. Following is the structure of a clause.

```
<query> <operator> [query|value literal] [message] [or|OR]
```

Keep in mind the following key points from Writing AWS CloudFormation Guard rules when you work with filters:

- Combine clauses by using Conjunctive Normal Form (CNF).
- Specify each conjunction (and) clause on a new line.
- Specify disjunctions (`or`) by using the `or` keyword between two clauses.

The following example demonstrates conjunctive and disjunctive clauses.

```
resourceType == 'AWS::EC2::SecurityGroup'
InputParameters.TcpBlockedPorts not empty
```

```
InputParameters.TcpBlockedPorts[*] {
    this in r(100, 400] or
    this in r(4000, 65535]
}
```

Using clauses for selection criteria

You can apply filtering to any collection. Filtering can be applied directly on attributes in the input that are already a collection like `securityGroups: [....]`. You can also apply filtering against a query, which is always a collection of values. You can use all features of clauses, including conjunctive normal form, for filtering.

The following common query is often used when selecting resources by type from a CloudFormation template.

```
Resources.*[ Type == 'AWS::IAM::Role' ]
```

The query `Resources.*` returns all values present in the `Resources` section of the input. For the example template input in Defining queries, the query returns the following.

```
- Type: AWS::IAM::Role
  ...
- Type: AWS::EC2::Instance
  ...
- Type: AWS::EC2::VPC
  ...
- Type: AWS::EC2::Subnet
  ...
- Type: AWS::EC2::Subnet
  ...
```

Now, apply the filter against this collection. The criterion to match is `Type == AWS::IAM::Role`. Following is the output of the query after the filter is applied.

```
- Type: AWS::IAM::Role
  ...
```

Next, check various clauses for `AWS::IAM::Role` resources.

```
let all_resources = Resources.*
```

```
let all_iam_roles = %all_resources[ Type == 'AWS::IAM::Role' ]
```

The following is an example filtering query that selects all `AWS::IAM::Policy` and `AWS::IAM::ManagedPolicy` resources.

```
Resources.*[
    Type in [ /IAM::Policy/,
              /IAM::ManagedPolicy/ ]
]
```

The following example checks if these policy resources have a `PolicyDocument` specified.

```
Resources.*[
    Type in [ /IAM::Policy/,
              /IAM::ManagedPolicy/ ]
    Properties.PolicyDocument exists
]
```

Building out more complex filtering needs

Consider the following example of an AWS Config configuration item for ingress and egress security groups information.

```
---
resourceType: 'AWS::EC2::SecurityGroup'
configuration:
  ipPermissions:
    - fromPort: 172
      ipProtocol: tcp
      toPort: 172
      ipv4Ranges:
        - cidrIp: 10.0.0.0/24
        - cidrIp: 0.0.0.0/0
    - fromPort: 89
      ipProtocol: tcp
      ipv6Ranges:
        - cidrIpv6: '::/0'
      toPort: 189
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: 1.1.1.1/32
    - fromPort: 89
```

```
      ipProtocol: '-1'
      toPort: 189
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: 1.1.1.1/32
  ipPermissionsEgress:
    - ipProtocol: '-1'
      ipv6Ranges: []
      prefixListIds: []
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: 0.0.0.0/0
      ipRanges:
        - 0.0.0.0/0
  tags:
    - key: Name
      value: good-sg-delete-me
  vpcId: vpc-0123abcd
InputParameter:
  TcpBlockedPorts:
    - 3389
    - 20
    - 21
    - 110
    - 143
```

Note the following:

- `ipPermissions` (ingress rules) is a collection of rules inside a configuration block.
- Each rule structure contains attributes such as `ipv4Ranges` and `ipv6Ranges` to specify a collection of CIDR blocks.

Let’s write a rule that selects any ingress rules that allow connections from any IP address, and verifies that the rules do not allow TCP blocked ports to be exposed.

Start with the query portion that covers IPv4, as shown in the following example.

```
configuration.ipPermissions[
    #
    # at least one ipv4Ranges equals ANY IPv4
    #
    some ipv4Ranges[*].cidrIp == '0.0.0.0/0'
```

```
]
```

The some keyword is useful in this context. All queries return a collection of values that match the query. By default, Guard evaluates that all values returned as a result of the query are matched against checks. However, this behavior might not always be what you need for checks. Consider the following part of the input from the configuration item.

```
ipv4Ranges:
  - cidrIp: 10.0.0.0/24
  - cidrIp: 0.0.0.0/0 # any IP allowed
```

There are two values present for `ipv4Ranges`. Not all `ipv4Ranges` values equal an IP address denoted by `0.0.0.0/0`. You want to see if at least one value matches `0.0.0.0/0`. You tell Guard that not all results returned from a query need to match, but at least one result must match. The `some` keyword tells Guard to ensure that one or more values from the resultant query match the check. If no query result values match, Guard throws an error.

Next, add IPv6, as shown in the following example.

```
configuration.ipPermissions[
    #
    # at-least-one ipv4Ranges equals ANY IPv4
    #
    some ipv4Ranges[*].cidrIp == '0.0.0.0/0' or
    #
    # at-least-one ipv6Ranges contains ANY IPv6
    #
    some ipv6Ranges[*].cidrIpv6 == '::/0'
]
```

Finally, in the following example, validate that the protocol is not udp.

```
configuration.ipPermissions[
    #
    # at-least-one ipv4Ranges equals ANY IPv4
    #
    some ipv4Ranges[*].cidrIp == '0.0.0.0/0' or
    #
    # at-least-one ipv6Ranges contains ANY IPv6
    #
    some ipv6Ranges[*].cidrIpv6 == '::/0'
```

```
    #
    # and ipProtocol is not udp
    #
    ipProtocol != 'udp' ]
]
```

The following is the complete rule.

```
rule any_ip_ingress_checks
{

    let ports = InputParameter.TcpBlockedPorts[*]

    let targets = configuration.ipPermissions[
        #
        # if either ipv4 or ipv6 that allows access from any address
        #
        some ipv4Ranges[*].cidrIp == '0.0.0.0/0' or
        some ipv6Ranges[*].cidrIpv6 == '::/0'

        #
        # the ipProtocol is not UDP
        #
        ipProtocol != 'udp' ]

    when %targets !empty
    {
        %targets {
            ipProtocol != '-1'
            <<
              result: NON_COMPLIANT
              check_id: HUB_ID_2334
              message: Any IP Protocol is allowed
            >>

            when fromPort exists
                 toPort exists
            {
                let each_target = this
                %ports {
                    this < %each_target.fromPort or
                    this > %each_target.toPort
```

```
                    <<
                        result: NON_COMPLIANT
                        check_id: HUB_ID_2340
                        message: Blocked TCP port was allowed in range
                    >>
                }
            }

        }
    }
}
```

Separating collections based on their contained types

When using infrastructure as code (IaC) configuration templates, you might encounter a collection that contains references to other entities within the configuration template. The following is an example CloudFormation template that describes Amazon Elastic Container Service (Amazon ECS) tasks with a local reference to `TaskRoleArn`, a reference to `TaskArn`, and a direct string reference.

```
Parameters:
  TaskArn:
    Type: String
Resources:
  ecsTask:
    Type: 'AWS::ECS::TaskDefinition'
    Metadata:
      SharedExectionRole: allowed
    Properties:
      TaskRoleArn: 'arn:aws:....'
      ExecutionRoleArn: 'arn:aws:...'
  ecsTask2:
    Type: 'AWS::ECS::TaskDefinition'
    Metadata:
      SharedExectionRole: allowed
    Properties:
      TaskRoleArn:
        'Fn::GetAtt':
          - iamRole
          - Arn
      ExecutionRoleArn: 'arn:aws:...2'
  ecsTask3:
    Type: 'AWS::ECS::TaskDefinition'
```

```
    Metadata:
      SharedExectionRole: allowed
    Properties:
      TaskRoleArn:
        Ref: TaskArn
      ExecutionRoleArn: 'arn:aws:...2'
  iamRole:
    Type: 'AWS::IAM::Role'
    Properties:
      PermissionsBoundary: 'arn:aws:...3'
```

Consider the following query.

```
let ecs_tasks = Resources.*[ Type == 'AWS::ECS::TaskDefinition' ]
```

This query returns a collection of values that contains all three `AWS::ECS::TaskDefinition` resources shown in the example template. Separate `ecs_tasks` that contain `TaskRoleArn` local references from others, as shown in the following example.

```
let ecs_tasks = Resources.*[ Type == 'AWS::ECS::TaskDefinition' ]

let ecs_tasks_role_direct_strings = %ecs_tasks[
    Properties.TaskRoleArn is_string ]

let ecs_tasks_param_reference = %ecs_tasks[
    Properties.TaskRoleArn.'Ref' exists ]

rule task_role_from_parameter_or_string {
    %ecs_tasks_role_direct_strings !empty or
    %ecs_tasks_param_reference !empty
}

rule disallow_non_local_references {
    # Known issue for rule access: Custom message must start on the same line
    not task_role_from_parameter_or_string
    <<
        result: NON_COMPLIANT
        message: Task roles are not local to stack definition
    >>
}
```

Assigning and referencing variables in Guard rules

You can assign variables in your AWS CloudFormation Guard rules files to store information that you want to reference in your Guard rules. Guard supports one-shot variable assignment. Variables are evaluated lazily, meaning that Guard only evaluates variables when rules are run.

Topics

- Assigning variables
- Referencing variables
- Variable scope
- Examples of variables in Guard rules files

Assigning variables

Use the `let` keyword to initialize and assign a variable. As a best practice, use snake case for variable names. Variables can store static literals or dynamic properties resulting from queries. In the following example, the variable `ecs_task_definition_task_role_arn` stores the static string value `arn:aws:iam:123456789012:role/my-role-name`.

```
let ecs_task_definition_task_role_arn = 'arn:aws:iam::123456789012:role/my-role-name'
```

In the following example, the variable `ecs_tasks` stores the results of a query that searches for all `AWS::ECS::TaskDefinition` resources in an CloudFormation template. You could reference `ecs_tasks` to access information about those resources when you write rules.

```
let ecs_tasks = Resources.*[
    Type == 'AWS::ECS::TaskDefinition'
]
```

Referencing variables

Use the `%` prefix to reference a variable.

Based on the `ecs_task_definition_task_role_arn` variable example in Assigning variables, you can reference `ecs_task_definition_task_role_arn` in the `query|value literal` section of a Guard rule clause. Using that reference ensures that the value specified for the TaskDefinitionArn property of any AWS::ECS::TaskDefinition resources in a CloudFormation template is the static string value arn:aws:iam:123456789012:role/my-role-name.

```
Resources.*.Properties.TaskDefinitionArn == %ecs_task_definition_role_arn
```

Based on the ecs_tasks variable example in Assigning variables, you can reference ecs_tasks in a query (for example, %ecs_tasks.Properties). First, Guard evaluates the variable ecs_tasks and then uses the returned values to traverse the hierarchy. If the variable ecs_tasks resolves to non-string values, then Guard throws an error.

> **Note**
>
> Currently, Guard doesn't support referencing variables inside custom error messages.

Variable scope

Scope refers to the visibility of variables defined in a rules file. A variable name can only be used once within a scope. There are three levels where a variable can be declared, or three possible variable scopes:

- **File-level** – Usually declared at the top of the rules file, you can use file-level variables in all rules within the rules file. They are visible to the entire file.

  In the following example rules file, the variables ecs_task_definition_task_role_arn and ecs_task_definition_execution_role_arn are initialized at the file-level.

```
let ecs_task_definition_task_role_arn = 'arn:aws:iam::123456789012:role/my-task-role-name'
let ecs_task_definition_execution_role_arn = 'arn:aws:iam::123456789012:role/my-execution-role-name'

rule check_ecs_task_definition_task_role_arn
{
    Resources.*.Properties.TaskRoleArn == %ecs_task_definition_task_role_arn
}

rule check_ecs_task_definition_execution_role_arn
{
```

```
    Resources.*.Properties.ExecutionRoleArn ==
 %ecs_task_definition_execution_role_arn
}
```

- **Rule-level** – Declared within a rule, rule-level variables are only visible to that specific rule. Any references outside of the rule result in an error.

  In the following example rules file, the variables `ecs_task_definition_task_role_arn` and `ecs_task_definition_execution_role_arn` are initialized at the rule-level. The `ecs_task_definition_task_role_arn` can only be referenced within the `check_ecs_task_definition_task_role_arn` named rule. You can only reference the `ecs_task_definition_execution_role_arn` variable within the `check_ecs_task_definition_execution_role_arn` named rule.

```
rule check_ecs_task_definition_task_role_arn
{
    let ecs_task_definition_task_role_arn = 'arn:aws:iam::123456789012:role/my-task-
role-name'
    Resources.*.Properties.TaskRoleArn == %ecs_task_definition_task_role_arn
}

rule check_ecs_task_definition_execution_role_arn
{
    let ecs_task_definition_execution_role_arn = 'arn:aws:iam::123456789012:role/my-
execution-role-name'
    Resources.*.Properties.ExecutionRoleArn ==
 %ecs_task_definition_execution_role_arn
}
```

- **Block-level** – Declared within a block, such as a `when` clause, block-level variables are only visible to that specific block. Any references outside of the block result in an error.

  In the following example rules file, the variables `ecs_task_definition_task_role_arn` and `ecs_task_definition_execution_role_arn` are initialized at the block-level within the `AWS::ECS::TaskDefinition` type block. You can only reference the `ecs_task_definition_task_role_arn` and `ecs_task_definition_execution_role_arn` variables within the `AWS::ECS::TaskDefinition` type blocks for their respective rules.

```
rule check_ecs_task_definition_task_role_arn
{
```

```
    AWS::ECS::TaskDefinition
    {
        let ecs_task_definition_task_role_arn = 'arn:aws:iam::123456789012:role/my-
task-role-name'
        Properties.TaskRoleArn == %ecs_task_definition_task_role_arn
    }
}

rule check_ecs_task_definition_execution_role_arn
{
    AWS::ECS::TaskDefinition
    {
        let ecs_task_definition_execution_role_arn = 'arn:aws:iam::123456789012:role/
my-execution-role-name'
        Properties.ExecutionRoleArn == %ecs_task_definition_execution_role_arn
    }
}
```

Examples of variables in Guard rules files

The following sections provide examples of both static and dynamic assignment of variables.

Static assignment

The following is an example CloudFormation template.

```
Resources:
  EcsTask:
    Type: 'AWS::ECS::TaskDefinition'
    Properties:
      TaskRoleArn: 'arn:aws:iam::123456789012:role/my-role-name'
```

Based on this template, you can write a rule called `check_ecs_task_definition_task_role_arn` that ensures that the `TaskRoleArn` property of all `AWS::ECS::TaskDefinition` template resources is `arn:aws:iam::123456789012:role/my-role-name`.

```
rule check_ecs_task_definition_task_role_arn
{
    let ecs_task_definition_task_role_arn = 'arn:aws:iam::123456789012:role/my-role-
name'
```

```
    Resources.*.Properties.TaskRoleArn == %ecs_task_definition_task_role_arn
}
```

Within the scope of the rule, you can initialize a variable called `ecs_task_definition_task_role_arn` and assign to it the static string value `'arn:aws:iam::123456789012:role/my-role-name'`. The rule clause checks whether the value specified for the `TaskRoleArn` property of the `EcsTask` resource is `arn:aws:iam::123456789012:role/my-role-name` by referencing the `ecs_task_definition_task_role_arn` variable in the `query|value literal` section.

Dynamic assignment

The following is an example CloudFormation template.

```
Resources:
  EcsTask:
    Type: 'AWS::ECS::TaskDefinition'
    Properties:
      TaskRoleArn: 'arn:aws:iam::123456789012:role/my-role-name'
```

Based on this template, you can initialize a variable called `ecs_tasks` within the scope of the file and assign to it the query `Resources.*[ Type == 'AWS::ECS::TaskDefinition'`. Guard queries all resources in the input template and stores information about them in `ecs_tasks`. You can also write a rule called `check_ecs_task_definition_task_role_arn` that ensures that the `TaskRoleArn` property of all `AWS::ECS::TaskDefinition` template resources is `arn:aws:iam::123456789012:role/my-role-name`

```
let ecs_tasks = Resources.*[
    Type == 'AWS::ECS::TaskDefinition'
]

rule check_ecs_task_definition_task_role_arn
{
    %ecs_tasks.Properties.TaskRoleArn == 'arn:aws:iam::123456789012:role/my-role-name'
}
```

The rule clause checks whether the value specified for the `TaskRoleArn` property of the `EcsTask` resource is `arn:aws:iam::123456789012:role/my-role-name` by referencing the `ecs_task_definition_task_role_arn` variable in the `query` section.

Enforcing CloudFormation template configuration

Let's walk through a more complex example of a production use case. In this example, we write Guard rules to ensure stricter controls on how Amazon ECS tasks are defined.

The following is an example CloudFormation template.

```
Resources:
  EcsTask:
    Type: 'AWS::ECS::TaskDefinition'
    Properties:
      TaskRoleArn:
        'Fn::GetAtt': [TaskIamRole, Arn]
      ExecutionRoleArn:
        'Fn::GetAtt': [ExecutionIamRole, Arn]

  TaskIamRole:
    Type: 'AWS::IAM::Role'
    Properties:
      PermissionsBoundary: 'arn:aws:iam::123456789012:policy/MyExamplePolicy'

  ExecutionIamRole:
    Type: 'AWS::IAM::Role'
    Properties:
      PermissionsBoundary: 'arn:aws:iam::123456789012:policy/MyExamplePolicy'
```

Based on this template, we write the following rules to ensure that these requirements are met:

- Each `AWS::ECS::TaskDefinition` resource in the template has both a task role and an execution role attached.
- The task roles and execution roles are AWS Identity and Access Management (IAM) roles.
- The roles are defined in the template.
- The `PermissionsBoundary` property is specified for each role.

```
# Select all Amazon ECS task definition resources from the template
let ecs_tasks = Resources.*[
    Type == 'AWS::ECS::TaskDefinition'
]

# Select a subset of task definitions whose specified value for the TaskRoleArn
 property is an Fn::Gett-retrievable attribute
```

```
let task_role_refs = some %ecs_tasks.Properties.TaskRoleArn.'Fn::GetAtt'[0]

# Select a subset of TaskDefinitions whose specified value for the ExecutionRoleArn
 property is an Fn::Gett-retrievable attribute
let execution_role_refs = some %ecs_tasks.Properties.ExecutionRoleArn.'Fn::GetAtt'[0]

# Verify requirement #1
rule all_ecs_tasks_must_have_task_end_execution_roles
    when %ecs_tasks !empty
{
    %ecs_tasks.Properties {
        TaskRoleArn exists
        ExecutionRoleArn exists
    }
}

# Verify requirements #2 and #3
rule all_roles_are_local_and_type_IAM
    when all_ecs_tasks_must_have_task_end_execution_roles
{
    let task_iam_references = Resources.%task_role_refs
    let execution_iam_reference = Resources.%execution_role_refs

    when %task_iam_references !empty {
        %task_iam_references.Type == 'AWS::IAM::Role'
    }

    when %execution_iam_reference !empty {
        %execution_iam_reference.Type == 'AWS::IAM::Role'
    }
}

# Verify requirement #4
rule check_role_have_permissions_boundary
    when all_ecs_tasks_must_have_task_end_execution_roles
{
    let task_iam_references = Resources.%task_role_refs
    let execution_iam_reference = Resources.%execution_role_refs

    when %task_iam_references !empty {
        %task_iam_references.Properties.PermissionsBoundary exists
    }

    when %execution_iam_reference !empty {
```

```
        %execution_iam_reference.Properties.PermissionsBoundary exists
    }
}
```

Composing named-rule blocks in AWS CloudFormation Guard

When writing named-rule blocks using AWS CloudFormation Guard, you can use the following two styles of composition:

- Conditional dependency
- Correlational dependency

Using either of these styles of dependency composition helps promote reusability and reduces verbosity and repetition in named-rule blocks.

Topics

- Prerequisites
- Conditional dependency composition
- Correlational dependency composition

Prerequisites

Learn about named-rule blocks in Writing rules.

Conditional dependency composition

In this style of composition, the evaluation of a `when` block or a named-rule block has a conditional dependency on the evaluation result of one or more other named-rule blocks or clauses. The following example Guard rules file contains named-rule blocks that demonstrate conditional dependencies.

```
# Named-rule block, rule_name_A
rule rule_name_A {
    Guard_rule_1
    Guard_rule_2
    ...
}
```

```
# Example-1, Named-rule block, rule_name_B, takes a conditional dependency on
 rule_name_A
rule rule_name_B when rule_name_A {
    Guard_rule_3
    Guard_rule_4
    ...
}

# Example-2, when block takes a conditional dependency on rule_name_A
when rule_name_A {
    Guard_rule_3
    Guard_rule_4
    ...
}

# Example-3, Named-rule block, rule_name_C, takes a conditional dependency on
 rule_name_A ^ rule_name_B
rule rule_name_C when rule_name_A
                      rule_name_B {
    Guard_rule_3
    Guard_rule_4
    ...
}

# Example-4, Named-rule block, rule_name_D, takes a conditional dependency on
 (rule_name_A v clause_A) ^ clause_B ^ rule_name_B
rule rule_name_D when rule_name_A OR
                      clause_A
                      clause_B
                      rule_name_B {
    Guard_rule_3
    Guard_rule_4
    ...
}
```

In the preceding example rules file, `Example-1` has the following possible outcomes:

- If `rule_name_A` evaluates to `PASS`, the Guard rules encapsulated by `rule_name_B` are evaluated.
- If `rule_name_A` evaluates to `FAIL`, the Guard rules encapsulated by `rule_name_B` are not evaluated. `rule_name_B` evaluates to `SKIP`.

- If `rule_name_A` evaluates to `SKIP`, the Guard rules encapsulated by `rule_name_B` are not evaluated. `rule_name_B` evaluates to `SKIP`.

> **Note**
>
> This case happens if `rule_name_A` conditionally depends on a rule that evaluates to `FAIL` and results in `rule_name_A` evaluating to `SKIP`.

Following is an example of a configuration management database (CMDB) configuration item from an AWS Config item for ingress and egress security groups information. This example demonstrates conditional dependency composition.

```
rule check_resource_type_and_parameter {
    resourceType == /AWS::EC2::SecurityGroup/
    InputParameters.TcpBlockedPorts NOT EMPTY
}

rule check_parameter_validity when check_resource_type_and_parameter {
    InputParameters.TcpBlockedPorts[*] {
        this in r[0,65535]
    }
}

rule check_ip_procotol_and_port_range_validity when check_parameter_validity {
    let ports = InputParameters.TcpBlockedPorts[*]

    #
    # select all ipPermission instances that can be reached by ANY IP address
    # IPv4 or IPv6 and not UDP
    #
    let configuration = configuration.ipPermissions[
        some ipv4Ranges[*].cidrIp == "0.0.0.0/0" or
        some ipv6Ranges[*].cidrIpv6 == "::/0"
        ipProtocol != 'udp' ]
    when %configuration !empty {
        %configuration {
            ipProtocol != '-1'

            when fromPort exists
                toPort exists {
                let ip_perm_block = this
```

```
                %ports {
                    this < %ip_perm_block.fromPort or
                    this > %ip_perm_block.toPort
                }
            }
        }
    }
}
```

In the preceding example, `check_parameter_validity` is conditionally dependent on `check_resource_type_and_parameter` and `check_ip_procotol_and_port_range_validity` is conditionally dependent on `check_parameter_validity`. The following is a configuration management database (CMDB) configuration item that conforms to the preceding rules.

```
---
version: '1.3'
resourceType: 'AWS::EC2::SecurityGroup'
resourceId: sg-12345678abcdefghi
configuration:
  description: Delete-me-after-testing
  groupName: good-sg-test-delete-me
  ipPermissions:
    - fromPort: 172
      ipProtocol: tcp
      ipv6Ranges: []
      prefixListIds: []
      toPort: 172
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: 0.0.0.0/0
      ipRanges:
        - 0.0.0.0/0
    - fromPort: 89
      ipProtocol: tcp
      ipv6Ranges:
        - cidrIpv6: '::/0'
      prefixListIds: []
      toPort: 89
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: 0.0.0.0/0
      ipRanges:
```

```
        - 0.0.0.0/0
  ipPermissionsEgress:
    - ipProtocol: '-1'
      ipv6Ranges: []
      prefixListIds: []
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: 0.0.0.0/0
      ipRanges:
        - 0.0.0.0/0
  tags:
    - key: Name
      value: good-sg-delete-me
  vpcId: vpc-0123abcd
InputParameters:
  TcpBlockedPorts:
    - 3389
    - 20
    - 110
    - 142
    - 1434
    - 5500
supplementaryConfiguration: {}
resourceTransitionStatus: None
```

Correlational dependency composition

In this style of composition, the evaluation of a `when` block or a named-rule block has a correlational dependency on the evaluation result of one or more other Guard rules. Correlational dependency can be achieved as follows.

```
# Named-rule block, rule_name_A, takes a correlational dependency on all of the Guard
 rules encapsulated by the named-rule block
rule rule_name_A {
    Guard_rule_1
    Guard_rule_2
    ...
}

# when block takes a correlational dependency on all of the Guard rules encapsulated by
 the when block
when condition {
    Guard_rule_1
```

```
    Guard_rule_2
    ...
}
```

To help you understand correlational dependency composition, review the following example of a Guard rules file.

```
#
# Allowed valid protocols for AWS::ElasticLoadBalancingV2::Listener resources
#
let allowed_protocols = [ "HTTPS", "TLS" ]

let elbs = Resources.*[ Type == 'AWS::ElasticLoadBalancingV2::Listener' ]

#
# If there are AWS::ElasticLoadBalancingV2::Listener resources present, ensure that
 they have protocols specified from the
# list of allowed protocols and that the Certificates property is not empty
#
rule ensure_all_elbs_are_secure when %elbs !empty {
    %elbs.Properties {
        Protocol in %allowed_protocols
        Certificates !empty
    }
}

#
# In addition to secure settings, ensure that AWS::ElasticLoadBalancingV2::Listener
 resources are private
#
rule ensure_elbs_are_internal_and_secure when %elbs !empty {
    ensure_all_elbs_are_secure
    %elbs.Properties.Scheme == 'internal'
}
```

In the preceding rules file, `ensure_elbs_are_internal_and_secure` has a correlational dependency on `ensure_all_elbs_are_secure`. The following is an example CloudFormation template that conforms to the preceding rules.

```
Resources:
  ServiceLBPublicListener46709EAA:
    Type: 'AWS::ElasticLoadBalancingV2::Listener'
```

```
    Properties:
      Scheme: internal
      Protocol: HTTPS
      Certificates:
        - CertificateArn: 'arn:aws:acm...'
  ServiceLBPublicListener4670GGG:
    Type: 'AWS::ElasticLoadBalancingV2::Listener'
    Properties:
      Scheme: internal
      Protocol: HTTPS
      Certificates:
        - CertificateArn: 'arn:aws:acm...'
```

Writing clauses to perform context-aware evaluations

AWS CloudFormation Guard clauses are evaluated against hierarchical data. The Guard evaluation engine resolves queries against incoming data by following hierarchical data as specified, using a simple dotted notation. Frequently, multiple clauses are needed to evaluate against a map of data or a collection. Guard provides a convenient syntax to write such clauses. The engine is contextually aware and uses the corresponding data associated for evaluations.

The following is an example of a Kubernetes Pod configuration with containers, to which you can apply context-aware evaluations.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: app
      image: 'images.my-company.example/app:v4'
      resources:
        requests:
          memory: 64Mi
          cpu: 0.25
        limits:
          memory: 128Mi
          cpu: 0.5
    - name: log-aggregator
      image: 'images.my-company.example/log-aggregator:v6'
      resources:
```

```
        requests:
          memory: 64Mi
          cpu: 0.25
        limits:
          memory: 128Mi
          cpu: 0.75
```

You can author Guard clauses to evaluate this data. When evaluating a rules file, the context is the entire input document. Following are example clauses that validate limits enforcement for containers specified in a Pod.

```
#
# At this level, the root document is available for evaluation
#

#
# Our rule only evaluates for apiVersion == v1 and K8s kind is Pod
#
rule ensure_container_limits_are_enforced
    when apiVersion == 'v1'
        kind == 'Pod'
{
    spec.containers[*] {
        resources.limits {
            #
            # Ensure that cpu attribute is set
            #
            cpu exists
            <<
                Id: K8S_REC_18
                Description: CPU limit must be set for the container
            >>

            #
            # Ensure that memory attribute is set
            #
            memory exists
            <<
                Id: K8S_REC_22
                Description: Memory limit must be set for the container
            >>
        }
    }
```

```
}
```

Understanding `context` in evaluations

At the rule-block level, the incoming context is the complete document. Evaluations for the `when` condition happen against this incoming root context where the `apiVersion` and `kind` attributes are located. In the previous example, these conditions evaluate to `true`.

Now, traverse the hierarchy in `spec.containers[*]` shown in the preceding example. For each traverse of the hierarchy, the context value changes accordingly. After the traversal of the `spec` block is finished, the context changes, as shown in the following example.

```
containers:
  - name: app
    image: 'images.my-company.example/app:v4'
    resources:
      requests:
        memory: 64Mi
        cpu: 0.25
      limits:
        memory: 128Mi
        cpu: 0.5
  - name: log-aggregator
    image: 'images.my-company.example/log-aggregator:v6'
    resources:
      requests:
        memory: 64Mi
        cpu: 0.25
      limits:
        memory: 128Mi
        cpu: 0.75
```

After traversing the `containers` attribute, the context is shown in the following example.

```
- name: app
  image: 'images.my-company.example/app:v4'
  resources:
    requests:
      memory: 64Mi
      cpu: 0.25
    limits:
      memory: 128Mi
```

```
        cpu: 0.5
  - name: log-aggregator
    image: 'images.my-company.example/log-aggregator:v6'
    resources:
      requests:
        memory: 64Mi
        cpu: 0.25
      limits:
        memory: 128Mi
        cpu: 0.75
```

Understanding loops

You can use the expression `[*]` to define a loop for all values contained in the array for the `containers` attribute. The block is evaluated for each element inside `containers`. In the preceding example rule snippet, the clauses contained inside the block define checks to be validated against a container definition. The block of clauses contained inside is evaluated twice, once for each container definition.

```
{
    spec.containers[*] {
       ...
    }
}
```

For each iteration, the context value is the value at that corresponding index.

> **Note**
>
> The only index access format supported is `[<integer>]` or `[*]`. Currently, Guard does not support ranges like `[2..4]`.

Arrays

Often in places where an array is accepted, single values are also accepted. For example, if there is only one container, the array can be dropped and the following input is accepted.

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: frontend
spec:
  containers:
    name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: 0.25
      limits:
        memory: "128Mi"
        cpu: 0.5
```

If an attribute can accept an array, ensure that your rule uses the array form. In the preceding example, you use `containers[*]` and not `containers`. Guard evaluates correctly when traversing the data when it encounters only the single-value form.

> **Note**
>
> Always use the array form when expressing access for a rule clause when an attribute accepts an array. Guard evaluates correctly even in the case that a single value is used.

Using the form `spec.containers[*]` instead of `spec.containers`

Guard queries return a collection of resolved values. When you use the form `spec.containers`, the resolved values for the query contain the array referred to by `containers`, not the elements inside it. When you use the form `spec.containers[*]`, you refer to each individual element contained. Remember to use the `[*]` form whenever you intend to evaluate each element contained in the array.

Using `this` to reference the current context value

When you author a Guard rule, you can reference the context value by using `this`. Often, `this` is implicit because it's bound to the context's value. For example, `this.apiVersion`, `this.kind`, and `this.spec` are bound to the root or document. In contrast, `this.resources` is bound to each value for `containers`, such as `/spec/containers/0/` and `/spec/containers/1`. Similarly, `this.cpu` and `this.memory` map to limits, specifically `/spec/containers/0/resources/limits` and `/spec/containers/1/resources/limits`.

In the next example, the preceding rule for the Kubernetes Pod configuration is rewritten to use `this` explicitly.

```
rule ensure_container_limits_are_enforced
    when this.apiVersion == 'v1'
         this.kind == 'Pod'
{
    this.spec.containers[*] {
        this.resources.limits {
            #
            # Ensure that cpu attribute is set
            #
            this.cpu exists
            <<
                Id: K8S_REC_18
                Description: CPU limit must be set for the container
            >>

            #
            # Ensure that memory attribute is set
            #
            this.memory exists
            <<
                Id: K8S_REC_22
                Description: Memory limit must be set for the container
            >>
        }
    }
}
```

You don't need to use `this` explicitly. However, the `this` reference can be useful when working with scalars, as shown in the following example.

```
InputParameters.TcpBlockedPorts[*] {
    this in r[0, 65535)
    <<
        result: NON_COMPLIANT
        message: TcpBlockedPort not in range (0, 65535)
    >>
}
```

In the previous example, `this` is used to refer to each port number.

Potential errors with the usage of implicit `this`

When authoring rules and clauses, there are some common mistakes when referencing elements from the implicit `this` context value. For example, consider the following input datum to evaluate against (this must pass).

```
resourceType: 'AWS::EC2::SecurityGroup'
InputParameters:
  TcpBlockedPorts: [21, 22, 110]
configuration:
  ipPermissions:
  - fromPort: 172
    ipProtocol: tcp
    ipv6Ranges: []
    prefixListIds: []
    toPort: 172
    userIdGroupPairs: []
    ipv4Ranges:
      - cidrIp: "0.0.0.0/0"
  - fromPort: 89
    ipProtocol: tcp
    ipv6Ranges:
      - cidrIpv6: "::/0"
    prefixListIds: []
    toPort: 109
    userIdGroupPairs: []
    ipv4Ranges:
      - cidrIp: 10.2.0.0/24
```

When tested against the preceding template, the following rule results in an error because it makes an incorrect assumption of leveraging the implicit `this`.

```
rule check_ip_procotol_and_port_range_validity
{
    #
    # select all ipPermission instances that can be reached by ANY IP address
    # IPv4 or IPv6 and not UDP
    #
    let any_ip_permissions = configuration.ipPermissions[
        some ipv4Ranges[*].cidrIp == "0.0.0.0/0" or
        some ipv6Ranges[*].cidrIpv6 == "::/0"
```

```
        ipProtocol != 'udp' ]

    when %any_ip_permissions !empty
    {
        %any_ip_permissions {
            ipProtocol != '-1' # this here refers to each ipPermission instance
            InputParameters.TcpBlockedPorts[*] {
                fromPort > this or
                toPort   < this
                <<
                    result: NON_COMPLIANT
                    message: Blocked TCP port was allowed in range
                >>
            }
        }
    }
}
```

To walk through this example, save the preceding rules file with the name `any_ip_ingress_check.guard` and the data with the file name `ip_ingress.yaml`. Then, run the following `validate` command with these files.

```
cfn-guard validate -r any_ip_ingress_check.guard -d ip_ingress.yaml --show-clause-failures
```

In the following output, the engine indicates that its attempt to retrieve a property `InputParameters.TcpBlockedPorts[*]` on the value `/configuration/ipPermissions/0`, `/configuration/ipPermissions/1` failed.

```
Clause #2     FAIL(Block[Location[file:any_ip_ingress_check.guard, line:17,
 column:13]])

              Attempting to retrieve array index or key from map at Path = /
configuration/ipPermissions/0, Type was not an array/object map, Remaining Query =
 InputParameters.TcpBlockedPorts[*]

Clause #3     FAIL(Block[Location[file:any_ip_ingress_check.guard, line:17,
 column:13]])

              Attempting to retrieve array index or key from map at Path = /
configuration/ipPermissions/1, Type was not an array/object map, Remaining Query =
 InputParameters.TcpBlockedPorts[*]
```

To help understand this result, rewrite the rule using `this` explicitly referenced.

```
rule check_ip_procotol_and_port_range_validity
{
    #
    # select all ipPermission instances that can be reached by ANY IP address
    # IPv4 or IPv6 and not UDP
    #
    let any_ip_permissions = this.configuration.ipPermissions[
        some ipv4Ranges[*].cidrIp == "0.0.0.0/0" or
        some ipv6Ranges[*].cidrIpv6 == "::/0"

        ipProtocol != 'udp' ]

    when %any_ip_permissions !empty
    {
        %any_ip_permissions {
            this.ipProtocol != '-1' # this here refers to each ipPermission instance
            this.InputParameters.TcpBlockedPorts[*] {
                this.fromPort > this or
                this.toPort   < this
                <<
                    result: NON_COMPLIANT
                    message: Blocked TCP port was allowed in range
                >>
            }
        }
    }
}
```

`this.InputParameters` references each value contained inside the variable `any_ip_permissions`. The query assigned to the variable selects `configuration.ipPermissions` values that match. The error indicates an attempt to retrieve `InputParamaters` in this context, but `InputParameters` was in the root context.

The inner block also references variables that are out of scope, as shown in the following example.

```
{
    this.ipProtocol != '-1' # this here refers to each ipPermission instance
    this.InputParameter.TcpBlockedPorts[*] { # ERROR referencing InputParameter off /
configuration/ipPermissions[*]
        this.fromPort > this or # ERROR: implicit this refers to values inside /
InputParameter/TcpBlockedPorts[*]
```

```
        this.toPort   < this
        <<
            result: NON_COMPLIANT
            message: Blocked TCP port was allowed in range
        >>
    }
}
```

`this` refers to each port value in `[21, 22, 110]`, but it also refers to `fromPort` and `toPort`. They both belong to the outer block scope.

Resolving errors with the implicit use of `this`

Use variables to explicitly assign and reference values. First, `InputParameter.TcpBlockedPorts` is part of the input (root) context. Move `InputParameter.TcpBlockedPorts` out of the inner block and assign it explicitly, as shown in the following example.

```
rule check_ip_procotol_and_port_range_validity
{
     let ports = InputParameters.TcpBlockedPorts[*]
    # ... cut off for illustrating change
}
```

Then, refer to this variable explicitly.

```
rule check_ip_procotol_and_port_range_validity
{
    #
    # Important: Assigning InputParameters.TcpBlockedPorts results in an ERROR.
    # We need to extract each port inside the array. The difference is the query
    # InputParameters.TcpBlockedPorts returns [[21, 20, 110]] whereas the query
    # InputParameters.TcpBlockedPorts[*] returns [21, 20, 110].
    #
    let ports = InputParameters.TcpBlockedPorts[*]

    #
    # select all ipPermission instances that can be reached by ANY IP address
    # IPv4 or IPv6 and not UDP
    #
    let any_ip_permissions = configuration.ipPermissions[
        some ipv4Ranges[*].cidrIp == "0.0.0.0/0" or
```

```
        some ipv6Ranges[*].cidrIpv6 == "::/0"

        ipProtocol != 'udp' ]

    when %any_ip_permissions !empty
    {
        %any_ip_permissions {
            this.ipProtocol != '-1' # this here refers to each ipPermission instance
            %ports {
                this.fromPort > this or
                this.toPort   < this
                <<
                    result: NON_COMPLIANT
                    message: Blocked TCP port was allowed in range
                >>
            }
        }
    }
}
```

Do the same for inner `this` references within `%ports`.

However, all errors aren't fixed yet because the loop inside `ports` still has an incorrect reference. The following example shows the removal of the incorrect reference.

```
rule check_ip_procotol_and_port_range_validity
{
    #
    # Important: Assigning InputParameters.TcpBlockedPorts results in an ERROR.
    # We need to extract each port inside the array. The difference is the query
    # InputParameters.TcpBlockedPorts returns [[21, 20, 110]] whereas the query
    # InputParameters.TcpBlockedPorts[*] returns [21, 20, 110].
    #
    let ports = InputParameters.TcpBlockedPorts[*]

    #
    # select all ipPermission instances that can be reached by ANY IP address
    # IPv4 or IPv6 and not UDP
    #
    let any_ip_permissions = configuration.ipPermissions[
        #
        # if either ipv4 or ipv6 that allows access from any address
        #
```

```
        some ipv4Ranges[*].cidrIp == '0.0.0.0/0' or
        some ipv6Ranges[*].cidrIpv6 == '::/0'

        #
        # the ipProtocol is not UDP
        #
        ipProtocol != 'udp' ]

    when %any_ip_permissions !empty
    {
        %any_ip_permissions {
            ipProtocol != '-1'
            <<
              result: NON_COMPLIANT
              check_id: HUB_ID_2334
              message: Any IP Protocol is allowed
            >>

            when fromPort exists
                 toPort exists
            {
                let each_any_ip_perm = this
                %ports {
                    this < %each_any_ip_perm.fromPort or
                    this > %each_any_ip_perm.toPort
                    <<
                        result: NON_COMPLIANT
                        check_id: HUB_ID_2340
                        message: Blocked TCP port was allowed in range
                    >>
                }
            }
        }
    }
}
```

Next, run the `validate` command again. This time, it passes.

```
cfn-guard validate -r any_ip_ingress_check.guard -d ip_ingress.yaml --show-clause-
failures
```

The following is the output of the `validate` command.

```
ip_ingress.yaml Status = PASS
PASS rules
check_ip_procotol_and_port_range_validity    PASS
```

To test this approach for failures, the following example uses a payload change.

```
resourceType: 'AWS::EC2::SecurityGroup'
InputParameters:
  TcpBlockedPorts: [21, 22, 90, 110]
configuration:
  ipPermissions:
    - fromPort: 172
      ipProtocol: tcp
      ipv6Ranges: []
      prefixListIds: []
      toPort: 172
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: "0.0.0.0/0"
    - fromPort: 89
      ipProtocol: tcp
      ipv6Ranges:
        - cidrIpv6: "::/0"
      prefixListIds: []
      toPort: 109
      userIdGroupPairs: []
      ipv4Ranges:
        - cidrIp: 10.2.0.0/24
```

90 is within the range from 89–109 that has any IPv6 address allowed. The following is the output of the `validate` command after running it again.

```
Clause #3           FAIL(Clause(Location[file:any_ip_ingress_check.guard, line:43,
 column:21], Check: _  LESS THAN %each_any_ip_perm.fromPort))
                    Comparing Int((Path("/InputParameters/TcpBlockedPorts/2"), 90))
 with Int((Path("/configuration/ipPermissions/1/fromPort"), 89)) failed
                    (DEFAULT: NO_MESSAGE)
Clause #4           FAIL(Clause(Location[file:any_ip_ingress_check.guard, line:44,
 column:21], Check: _  GREATER THAN %each_any_ip_perm.toPort))
                    Comparing Int((Path("/InputParameters/TcpBlockedPorts/2"), 90))
 with Int((Path("/configuration/ipPermissions/1/toPort"), 109)) failed
```

```
                                                  result: NON_COMPLIANT
                                                  check_id: HUB_ID_2340
                                                  message: Blocked TCP port was allowed in
 range
```

Testing AWS CloudFormation Guard rules

You can use the AWS CloudFormation Guard built-in unit testing framework to verify that your Guard rules work as intended. This section provides a walkthrough on how to write a unit testing file and how to use it to test your rules file with the `test` command.

Your unit test file must have one of the following extensions: `.json`, `.JSON`, `.jsn`, `.yaml`, `.YAML`, or `.yml`.

Topics

- Prerequisites
- Overview of Guard unit testing files
- Walkthrough of writing a Guard rules unit testing file

Prerequisites

Write Guard rules to evaluate your input data against. For more information, see Writing Guard rules.

Overview of Guard unit testing files

Guard unit testing files are JSON- or YAML-formatted files that contain multiple inputs and the expected outcomes for rules written inside a Guard rules file. There can be multiple samples to assess different expectations. We recommend that you start by testing for empty inputs and then progressively add information for assessing various rules and clauses.

Also, we recommend that you name unit testing files using the suffix `_test.json` or `_tests.yaml`. For example, if you have a rules file named `my_rules.guard`, name your unit testing file `my_rules_tests.yaml`.

Syntax

The following shows the syntax of a unit testing file in YAML format.

```
---
- name: <TEST NAME>
  input:
     <SAMPLE INPUT>
   expectations:
     rules:
       <RULE NAME>: [PASS|FAIL|SKIP]
```

Properties

Following are the properties of a Guard test file.

`input`

Data to test your rules against. We recommend that your first test uses an empty input, as shown in the following example.

```
---
- name: MyTest1
  input {}
```

For subsequent tests, add input data to test.

Required: Yes

`expectations`

The expected outcome when specific rules are evaluated against your input data. Specify one or multiple rules that you want to test in addition to the expected outcome for each rule. The expected outcome must be one of the following:

- `PASS` – When run against your input data, the rules evaluate to `true`.
- `FAIL` – When run against your input data, the rules evaluate to `false`.
- `SKIP` – When run against your input data, the rule isn't triggered.

```
expectations:
    rules:
      check_rest_api_is_private: PASS
```

Required: Yes

Walkthrough of writing a Guard rules unit testing file

The following is a rules file named `api_gateway_private.guard`. The intent for this rule is to check whether all Amazon API Gateway resource types defined in a CloudFormation template are deployed for private access only. It also checks whether at least one policy statement allows access from a virtual private cloud (VPC).

```
#
# Select all AWS::ApiGateway::RestApi resources
#     present in the Resources section of the template.
#
let api_gws = Resources.*[ Type == 'AWS::ApiGateway::RestApi']

#
# Rule intent:
# 1) All AWS::ApiGateway::RestApi resources deployed must be private.

# 2) All AWS::ApiGateway::RestApi resources deployed must have at least one AWS
 Identity and Access Management (IAM) policy condition key to allow access from a VPC.
#
# Expectations:
# 1) SKIP when there are no AWS::ApiGateway::RestApi resources in the template.
# 2) PASS when:
#     ALL AWS::ApiGateway::RestApi resources in the template have
 the EndpointConfiguration property set to Type: PRIVATE.
#     ALL AWS::ApiGateway::RestApi resources in the template have one IAM condition key
 specified in the Policy property with aws:sourceVpc or :SourceVpc.
# 3) FAIL otherwise.

#
#

rule check_rest_api_is_private when %api_gws !empty {
    %api_gws {
        Properties.EndpointConfiguration.Types[*] == "PRIVATE"

    }
}

rule check_rest_api_has_vpc_access when check_rest_api_is_private {
    %api_gws {
        Properties {
            #
```

```
            # ALL AWS::ApiGateway::RestApi resources in the template have one IAM
 condition key specified in the Policy property with
            #     aws:sourceVpc or :SourceVpc
            #
            some Policy.Statement[*] {
                Condition.*[ keys == /aws:[sS]ource(Vpc|VPC|Vpce|VPCE)/ ] !empty
            }
        }
    }
}
```

This walkthrough tests the first rule intent: All `AWS::ApiGateway::RestApi` resources deployed must be private.

1. Create a unit testing file called `api_gateway_private_tests.yaml` that contains the following initial test. With the initial test, add an empty input and expect that the rule `check_rest_api_is_private` will skip because there are no `AWS::ApiGateway::RestApi` resources as inputs.

```
---
- name: MyTest1
  input: {}
  expectations:
    rules:
      check_rest_api_is_private: SKIP
```

2. Run the first test in your terminal using the `test` command. For the `--rules-file` parameter, specify your rules file. For the `--test-data` parameter, specify your unit testing file.

```
cfn-guard test --rules-file api_gateway_private.guard --test-data
 api_gateway_private_tests.yaml
```

   The outcome for the first test is `PASS`.

```
Test Case #1
Name: "MyTest1"
  PASS Rules:
    check_rest_api_is_private: Expected = SKIP, Evaluated = SKIP
```

3. Add another test to your unit testing file. Now, extend the testing to include empty resources. The following is the updated `api_gateway_private_tests.yaml` file.

```
---
- name: MyTest1
  input: {}
  expectations:
    rules:
      check_rest_api_is_private: SKIP
- name: MyTest2
  input:
     Resources: {}
  expectations:
    rules:
      check_rest_api_is_private: SKIP
```

4. Run `test` with the updated unit testing file.

```
cfn-guard test --rules-file api_gateway_private.guard --test-data
 api_gateway_private_tests.yaml
```

The outcome for the second test is `PASS`.

```
Test Case #1
Name: "MyTest1"
  PASS Rules:
    check_rest_api_is_private: Expected = SKIP, Evaluated = SKIP
Test Case #2
Name: "MyTest2"
  PASS Rules:
    check_rest_api_is_private: Expected = SKIP, Evaluated = SKIP
```

5. Add two more tests to your unit testing file. Extend the testing to include the following:

- An `AWS::ApiGateway::RestApi` resource with no properties specified.

> **Note**
>
> This isn’t a valid CloudFormation template, but it's useful to test whether the rule works correctly even for malformed inputs.

Expect that this test will fail because the `EndpointConfiguration` property isn't specified and is therefore not set to `PRIVATE`.

- An `AWS::ApiGateway::RestApi` resource that satisfies the first intent with the `EndpointConfiguration` property set to `PRIVATE` but does not satisfy the second intent because it has no policy statements defined. Expect that this test will pass.

The following is the updated unit testing file.

```
---
- name: MyTest1
  input: {}
  expectations:
    rules:
      check_rest_api_is_private: SKIP
- name: MyTest2
  input:
     Resources: {}
  expectations:
    rules:
      check_rest_api_is_private: SKIP
- name: MyTest3
  input:
    Resources:
      apiGw:
        Type: AWS::ApiGateway::RestApi
  expectations:
    rules:
      check_rest_api_is_private: FAIL
- name: MyTest4
  input:
    Resources:
      apiGw:
        Type: AWS::ApiGateway::RestApi
        Properties:
          EndpointConfiguration:
            Types: "PRIVATE"
  expectations:
    rules:
      check_rest_api_is_private: PASS
```

6. Run `test` with the updated unit testing file.

```
cfn-guard test --rules-file api_gateway_private.guard --test-data
 api_gateway_private_tests.yaml \
```

The third outcome is `FAIL`, and the fourth outcome is `PASS`.

```
Test Case #1
Name: "MyTest1"
  PASS Rules:
    check_rest_api_is_private: Expected = SKIP, Evaluated = SKIP

Test Case #2
Name: "MyTest2"
  PASS Rules:
    check_rest_api_is_private: Expected = SKIP, Evaluated = SKIP

Test Case #3
Name: "MyTest3"
  PASS Rules:
    check_rest_api_is_private: Expected = FAIL, Evaluated = FAIL

Test Case #4
Name: "MyTest4"
  PASS Rules:
    check_rest_api_is_private: Expected = PASS, Evaluated = PASS
```

7. Comment out tests 1–3 in your unit testing file. Access the verbose context for the fourth test only. The following is the updated unit testing file.

```
---
#- name: MyTest1
#  input: {}
#  expectations:
#    rules:
#      check_rest_api_is_private_and_has_access: SKIP
#- name: MyTest2
#  input:
#     Resources: {}
#  expectations:
#    rules:
#      check_rest_api_is_private_and_has_access: SKIP
```

```
#- name: MyTest3
#  input:
#    Resources:
#      apiGw:
#        Type: AWS::ApiGateway::RestApi
#  expectations:
#    rules:
#      check_rest_api_is_private_and_has_access: FAIL
- name: MyTest4
  input:
    Resources:
      apiGw:
        Type: AWS::ApiGateway::RestApi
        Properties:
          EndpointConfiguration:
            Types: "PRIVATE"
  expectations:
    rules:
      check_rest_api_is_private: PASS
```

8. Inspect the evaluation results by running the `test` command in your terminal, using the `--verbose` flag. Verbose context is useful for understanding evaluations. In this case, it provides detailed information about why the fourth test succeeded with a PASS outcome.

```
cfn-guard test --rules-file api_gateway_private.guard --test-data
 api_gateway_private_tests.yaml \
  --verbose
```

Here is the output from that run.

```
Test Case #1
Name: "MyTest4"
  PASS Rules:
    check_rest_api_is_private: Expected = PASS, Evaluated = PASS
Rule(check_rest_api_is_private, PASS)
    |  Message: DEFAULT MESSAGE(PASS)
    Condition(check_rest_api_is_private, PASS)
        |  Message: DEFAULT MESSAGE(PASS)
        Clause(Clause(Location[file:api_gateway_private.guard, line:20, column:37],
 Check: %api_gws NOT EMPTY ), PASS)
            |  From: Map((Path("/Resources/apiGw"), MapValue { keys:
 [String((Path("/Resources/apiGw/Type"), "Type")), String((Path("/Resources/
```

```
apiGw/Properties"), "Properties"))], values: {"Type": String((Path("/Resources/
apiGw/Type"), "AWS::ApiGateway::RestApi")), "Properties": Map((Path("/
Resources/apiGw/Properties"), MapValue { keys: [String((Path("/Resources/
apiGw/Properties/EndpointConfiguration"), "EndpointConfiguration"))],
 values: {"EndpointConfiguration": Map((Path("/Resources/apiGw/Properties/
EndpointConfiguration"), MapValue { keys: [String((Path("/Resources/apiGw/
Properties/EndpointConfiguration/Types"), "Types"))], values: {"Types":
 String((Path("/Resources/apiGw/Properties/EndpointConfiguration/Types"),
 "PRIVATE"))} }))} }))} }))
            |  Message: (DEFAULT: NO_MESSAGE)
    Conjunction(cfn_guard::rules::exprs::GuardClause, PASS)
        |  Message: DEFAULT MESSAGE(PASS)
        Clause(Clause(Location[file:api_gateway_private.guard, line:22, column:5],
 Check: Properties.EndpointConfiguration.Types[*]  EQUALS String("PRIVATE")), PASS)
            |  Message: (DEFAULT: NO_MESSAGE)
```

The key observation from the output is the line `Clause(Location[file:api_gateway_private.guard, line:22, column:5], Check: Properties.EndpointConfiguration.Types[*] EQUALS String("PRIVATE")), PASS)`, which states that the check passed. The example also showed the case where `Types` was expected to be an array, but a single value was given. In that case, Guard continued to evaluate and provided a correct result.

9. Add a test case like the fourth test case to your unit testing file for an `AWS::ApiGateway::RestApi` resource with the `EndpointConfiguration` property specified. The test case will fail instead of pass. The following is the updated unit testing file.

```
---
#- name: MyTest1
#  input: {}
#  expectations:
#    rules:
#      check_rest_api_is_private_and_has_access: SKIP
#- name: MyTest2
#  input:
#     Resources: {}
#  expectations:
#    rules:
#      check_rest_api_is_private_and_has_access: SKIP
#- name: MyTest3
#  input:
#    Resources:
```

```
#      apiGw:
#        Type: AWS::ApiGateway::RestApi
#  expectations:
#    rules:
#      check_rest_api_is_private_and_has_access: FAIL
#- name: MyTest4
#  input:
#    Resources:
#      apiGw:
#        Type: AWS::ApiGateway::RestApi
#        Properties:
#          EndpointConfiguration:
#            Types: "PRIVATE"
#  expectations:
#    rules:
#      check_rest_api_is_private: PASS
- name: MyTest5
  input:
    Resources:
      apiGw:
        Type: AWS::ApiGateway::RestApi
        Properties:
          EndpointConfiguration:
            Types: [PRIVATE, REGIONAL]
  expectations:
    rules:
      check_rest_api_is_private: FAIL
```

10. Run the `test` command with the updated unit testing file using the `--verbose` flag.

```
cfn-guard test --rules-file api_gateway_private.guard --test-data
 api_gateway_private_tests.yaml \
 --verbose
```

The outcome is `FAIL` as expected because `REGIONAL` is specified for `EndpointConfiguration` but is not expected.

```
Test Case #1
Name: "MyTest5"
  PASS Rules:
    check_rest_api_is_private: Expected = FAIL, Evaluated = FAIL
Rule(check_rest_api_is_private, FAIL)
    |  Message: DEFAULT MESSAGE(FAIL)
```

```
    Condition(check_rest_api_is_private, PASS)
        |  Message: DEFAULT MESSAGE(PASS)
        Clause(Clause(Location[file:api_gateway_private.guard, line:20, column:37],
 Check: %api_gws NOT EMPTY ), PASS)
            |  From: Map((Path("/Resources/apiGw"), MapValue { keys:
 [String((Path("/Resources/apiGw/Type"), "Type")), String((Path("/Resources/
apiGw/Properties"), "Properties"))], values: {"Type": String((Path("/Resources/
apiGw/Type"), "AWS::ApiGateway::RestApi")), "Properties": Map((Path("/
Resources/apiGw/Properties"), MapValue { keys: [String((Path("/Resources/
apiGw/Properties/EndpointConfiguration"), "EndpointConfiguration"))],
 values: {"EndpointConfiguration": Map((Path("/Resources/apiGw/Properties/
EndpointConfiguration"), MapValue { keys: [String((Path("/Resources/apiGw/
Properties/EndpointConfiguration/Types"), "Types"))], values: {"Types":
 List((Path("/Resources/apiGw/Properties/EndpointConfiguration/Types"),
 [String((Path("/Resources/apiGw/Properties/EndpointConfiguration/Types/0"),
 "PRIVATE")), String((Path("/Resources/apiGw/Properties/EndpointConfiguration/
Types/1"), "REGIONAL"))]))} }))} }))} }))
            |  Message: DEFAULT MESSAGE(PASS)
    BlockClause(Block[Location[file:api_gateway_private.guard, line:21, column:3]],
 FAIL)
        |  Message: DEFAULT MESSAGE(FAIL)
        Conjunction(cfn_guard::rules::exprs::GuardClause, FAIL)
            |  Message: DEFAULT MESSAGE(FAIL)
            Clause(Clause(Location[file:api_gateway_private.guard, line:22,
 column:5], Check: Properties.EndpointConfiguration.Types[*]  EQUALS
 String("PRIVATE")), FAIL)
                |  From: String((Path("/Resources/apiGw/Properties/
EndpointConfiguration/Types/1"), "REGIONAL"))
                |  To: String((Path("api_gateway_private.guard/22/5/Clause/"),
 "PRIVATE"))
                |  Message: (DEFAULT: NO_MESSAGE)
```

The verbose output of the `test` command follows the structure of the rules file. Every block in the rules file is a block in the verbose output. The top-most block is each rule. If there are `when` conditions against the rule, they appear in a sibling condition block. In the following example, the condition `%api_gws !empty` is tested and it passes.

```
rule check_rest_api_is_private when %api_gws !empty {
```

Once the condition passes, we test the rule clauses.

```
%api_gws {
```

```
    Properties.EndpointConfiguration.Types[*] == "PRIVATE"
}
```

`%api_gws` is a block rule that corresponds to the `BlockClause` level in the output (line:21). The rule clauseis a set of conjunction (AND) clauses, where each conjunction clause is a set of disjunctions (ORs). The conjunction has a single clause, `Properties.EndpointConfiguration.Types[*] == "PRIVATE"`. Therefore, the verbose output shows a single clause. The path `/Resources/apiGw/Properties/EndpointConfiguration/Types/1` shows which values in the input are compared, which in this case is the element for `Types` indexed at 1.

In [Validating input data against Guard rules](), you can use the examples in this section to use the `validate` command to evaluate input data against rules.

Using input parameters with AWS CloudFormation Guard rules

AWS CloudFormation Guard allows you to use input parameters for dynamic data lookups during validation. This feature is particularly useful when you need to reference external data in your rules. However, when specifying input parameter keys, Guard requires that there are no conflicting paths.

How to use

1. Use the `--input-parameters` or `-i` flag to specify files containing input parameters. Multiple input parameter files can be specified and will be combined to form a common context. Input parameter keys can not have conflicting paths.
2. Use the `--data` or `-d` flag to specify the actual template file to be validated.

Example usage

1. Create an input parameter file (For example, `network.yaml`):

```
NETWORK:
  allowed_security_groups: ["sg-282850", "sg-292040"]
  allowed_prefix_lists: ["pl-63a5400a", "pl-02cd2c6b"]
```

2. Reference these parameters in your guard rule file (For example, `security_groups.guard`):

```
let groups = Resources.*[ Type == 'AWS::EC2::SecurityGroup' ]

let permitted_sgs = NETWORK.allowed_security_groups
let permitted_pls = NETWORK.allowed_prefix_lists
rule check_permitted_security_groups_or_prefix_lists(groups) {
    %groups {
        this in %permitted_sgs or
        this in %permitted_pls
    }
}

rule CHECK_PERMITTED_GROUPS when %groups !empty {
    check_permitted_security_groups_or_prefix_lists(
       %groups.Properties.GroupName
    )
}
```

3. Create a failing data template (For example, `security_groups_fail.yaml`):

```
# ---
# AWSTemplateFormatVersion: 2010-09-09
# Description: CloudFormation - EC2 Security Group

Resources:
  mySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupName: wrong
```

4. Run the validate command:

```
cfn-guard validate -r security_groups.guard -i network.yaml -d
 security_groups_fail.yaml
```

In this command:

- `-r` specifies the rule file.
- `-i` specifies the input parameter file.
- `-d` specifies the data file (template) to be validated.

Multiple input parameters

You can specify multiple input parameter files:

```
cfn-guard validate -r rules.guard -i params1.yaml -i params2.yaml -d template.yaml
```

All files specified with `-i` will be combined to form a single context for parameter lookup.

Validating input data against AWS CloudFormation Guard rules

You can use the AWS CloudFormation Guard `validate` command to validate data against Guard rules. For more information about the `validate` command, including its parameters and options, see validate.

Prerequisites

- Write Guard rules to validate your input data against. For more information, see Writing Guard rules.
- Test your rules to ensure that they work as intended. For more information, see Testing Guard rules.

Using the `validate` command

To validate your input data against your Guard rules, such as an AWS CloudFormation template, run the Guard `validate` command. For the `--rules` parameter, specify the name of a rules file. For the `--data` parameter, specify the name of the input data file.

```
cfn-guard validate --rules rules.guard --data template.json
```

If Guard successfully validates the templates, the `validate` command returns an exit status of `0` (`$?` in bash). If Guard identifies a rule violation, the `validate` command returns a status report of the rules that failed. Use the summary flag (`-s all`) to see the detailed evaluation tree that shows how Guard evaluated each rule.

```
template.json Status = FAIL
SKIP rules
rules.guard/aws_apigateway_deployment_checks    SKIP
rules.guard/aws_apigateway_stage_checks         SKIP
```

```
rules.guard/aws_dynamodb_table_checks           SKIP
PASS rules
rules.guard/aws_events_rule_checks              PASS
rules.guard/aws_iam_role_checks                 PASS
FAILED rules
rules.guard/aws_ec2_volume_checks               FAIL
rules.guard/mixed_types_checks                  FAIL
---
Evaluation of rules rules.guard against data template.json
--
Property [/Resources/vol2/Properties/Encrypted] in data [template.json] is not
 compliant with [rules.guard/aws_ec2_volume_checks] because provided value [false] did
 not match expected value [true]. Error Message []
Property traversed until [/Resources/vol2/Properties] in data [template.json] is not
 compliant with [rules.guard/aws_ec2_volume_checks] due to retrieval error. Error
 Message [Attempting to retrieve array index or key from map at path = /Resources/vol2/
Properties , Type was not an array/object map, Remaining Query = Size]
Property [/Resources/vol2/Properties/Encrypted] in data [template.json] is not
 compliant with [rules.guard/mixed_types_checks] because provided value [false] did not
 match expected value [true]. Error Message []
--
Rule [rules.guard/aws_iam_role_checks] is compliant for data [template.json]
Rule [rules.guard/aws_events_rule_checks] is compliant for data [template.json]
--
Rule [rules.guard/aws_apigateway_deployment_checks] is not applicable for data
 [template.json]
Rule [rules.guard/aws_apigateway_stage_checks] is not applicable for data
 [template.json]
Rule [rules.guard/aws_dynamodb_table_checks] is not applicable for data [template.json]
```

Validating multiple rules against multiple data files

To help maintain rules, you can write rules into multiple files and organize the rules as you want. Then, you can validate multiple rule files against a data file or multiple data files. The `validate` command can take a directory of files for the `--data` and `--rules` options. For example, you can run the following command where `/path/to/dataDirectory` contains one or more data files and `/path/to/ruleDirectory` contains one or more rules files.

```
cfn-guard validate --data /path/to/dataDirectory --rules /path/to/ruleDirectory
```

You can write rules to check whether various resources defined in multiple CloudFormation templates have the appropriate property assignments to guarantee encryption at rest. For search and maintenance ease, you can have rules for checking encryption at rest in each resource in separate files, called `s3_bucket_encryption.guard`, `ec2_volume_encryption.guard`, and `rds_dbinstance_encrytion.guard` in a directory with the path `~/GuardRules/encryption_at_rest`. The CloudFormation templates that you need to validate are in a directory with the path `~/CloudFormation/templates`. In this case, run the `validate` command as follows.

```
cfn-guard validate --data ~/CloudFormation/templates --rules ~/GuardRules/
encryption_at_rest
```

Troubleshooting AWS CloudFormation Guard

If you encounter issues while working with AWS CloudFormation Guard, consult the topics in this section.

Topics

- Clause fails when no resources of the selected type are present
- Guard does not evaluate CloudFormation template with short-form Fn::GetAtt references
- General troubleshooting topics

Clause fails when no resources of the selected type are present

When a query uses a filter like `Resources.*[ Type == 'AWS::ApiGateway::RestApi' ]`, if there are no `AWS::ApiGateway::RestApi` resources in the input, the clause evaluates to `FAIL`.

```
%api_gws.Properties.EndpointConfiguration.Types[*] == "PRIVATE"
```

To avoid this outcome, assign filters to variables and use the `when` condition check.

```
let api_gws = Resources.*[ Type == 'AWS::ApiGateway::RestApi' ]
    when %api_gws !empty { ...}
```

Guard does not evaluate CloudFormation template with short-form Fn::GetAtt references

Guard doesn't support the short forms of intrinsic functions. For example, using `!Join`, `!Sub` in a YAML-formatted CloudFormation template isn't supported. Instead, use the expanded forms of CloudFormation intrinsic functions. For example, use `Fn::Join`, `Fn::Sub` in YAML-formatted CloudFormation templates when evaluating them against Guard rules.

For more information about intrinsic functions, see the intrinsic function reference in the *AWS CloudFormation User Guide*.

General troubleshooting topics

- Verify that `string` literals don't contain embedded escaped strings. Guard doesn't support embedded escape strings in `string` literals. If your intent is to parse inline JSON strings, use the `json_parse()` function available in Guard 3.0.0 and later. For more information, see [Using built-in functions](#).
- Verify that your `!=` comparisons compare compatible data types. For example, a `string` and an `int` are not compatible data types for comparison. When performing `!=` comparison, if the values are incompatible, an error occurs internally. Currently, the error is suppressed and converted to `false` to satisfy the [PartialEq](#) trait in Rust.

AWS CloudFormation Guard CLI parameters and command reference

The following global parameters and commands are available through the AWS CloudFormation Guard command line interface (CLI).

Topics

- Guard CLI global parameters
- parse-tree
- rulegen
- test
- validate

Guard CLI global parameters

You can use the following parameters with any AWS CloudFormation Guard CLI command.

`-h`, `--help`

Prints help information.

`-V`, `--version`

Prints version information.

parse-tree

Generates a parse tree for the AWS CloudFormation Guard rules defined in a rules file.

Syntax

```
cfn-guard parse-tree
--output <value>
--rules <value>
```

Parameters

`-h`, `--help`

Prints help information.

`-p`, `--print-json`

Prints the output in JSON format.

`-y`, `--print-yaml`

Prints the output in YAML format.

`-V`, `--version`

Prints version information.

Options

`-o`, `--output`

Writes the generated tree to an output file.

`-r`, `--rules`

Provides a rules file.

Examples

```
cfn-guard parse-tree --output output.json --rules rules.guard
```

rulegen

Takes a JSON- or YAML-formatted AWS CloudFormation template file and autogenerates a set of AWS CloudFormation Guard rules that match the properties of the template resources. This command is a useful way to get started with rule writing or to create ready-to-use rules from known good templates.

Syntax

```
cfn-guard rulegen
```

```
--output <value>
--template <value>
```

Parameters

`-h`, `--help`

Prints help information.

`-V`, `--version`

Prints version information.

Options

`-o`, `--output`

Writes the generated rules to an output file. Given the potential for hundreds or even thousands of rules to emerge, we recommend using this option.

`-t`, `--template`

Provides the path to a CloudFormation template file in JSON or YAML format.

Examples

```
cfn-guard rulegen --output rules.guard --template template.json
```

test

Validates an AWS CloudFormation Guard rules file against a Guard unit testing file in JSON or YAML format to determine the success of individual rules.

Syntax

```
cfn-guard test
--rules-file <value>
--test-data <value>
```

Parameters

`-a`, `--alphabetical`

Sort alphabetically inside a directory.

`-h`, `--help`

Prints help information.

`-m`, `--last-modified`

Sorts by last-modified times within a directory

`-V`, `--version`

Prints version information.

`-v`, `--verbose`

Increases the output verbosity. Can be specified multiple times.

The verbose output follows the structure of the Guard rules file. Every block in the rules file is a block in the verbose output. The top-most block is each rule. If there are `when` conditions against the rule, they appear as a sibling condition block.

Options

`-d`, `--dir`

Provide the root directory for rules.

`-o`, `--output-format`

Specify the format in which the output should be displayed.

Default: `single-line-summary`

Allowed values: `json` | `yaml` | `single-line-summary` | `junit`

`-r`, `--rules-file`

Provides the name of a rules file.

`-t`, `--test-data`

Provides the name of a file or directory for data files in either JSON or YAML format.

Examples

```
cfn-guard test --rules-file rules.guard --test-data example.json
```

Output

```
PASS|FAIL Expected Rule = rule_name, Status = SKIP|FAIL|PASS, Got Status = SKIP|FAIL|
PASS
```

See also

[Testing Guard rules]

validate

Validates data against AWS CloudFormation Guard rules to determine success or failure.

Syntax

```
cfn-guard validate
--data <value>
--output-format <value>
--rules <value>
--show-summary <value>
--type <value>
```

Parameters

`-a`, `--alphabetical`

Validates files in a directory that is ordered alphabetically.

`-h`, `--help`

Prints help information.

`-m`, `--last-modified`

Validates files in a directory that is ordered by last-modified times.

`-P`, `--payload`

Provide rules and data in the following JSON format via `stdin`:

```
{"rules":["<rules 1>", "<rules 2>", ...], "data":["<data 1>", "<data 2>", ...]}
```

For example:

```
{"data": ["{\"Resources\":{\"NewVolume\":{\"Type\":\"AWS::EC2::Volume\",\"Properties
\":{\"Size\":500,\"Encrypted\":false,\"AvailabilityZone\":\"us-west-2b\"}},
\"NewVolume2\":{\"Type\":\"AWS::EC2::Volume\",\"Properties\":{\"Size\":50,\"Encrypted
\":false,\"AvailabilityZone\":\"us-west-2c\"}}},\"Parameters\":{\"InstanceName\":
\"TestInstance\"}}","{\"Resources\":{\"NewVolume\":{\"Type\":\"AWS::EC2::Volume\",
\"Properties\":{\"Size\":500,\"Encrypted\":false,\"AvailabilityZone\":\"us-west-2b\"}},
\"NewVolume2\":{\"Type\":\"AWS::EC2::Volume\",\"Properties\":{\"Size\":50,\"Encrypted
\":false,\"AvailabilityZone\":\"us-west-2c\"}}},\"Parameters\":{\"InstanceName
\":\"TestInstance\"}}"], "rules" : [ "Parameters.InstanceName == \"TestInstance
\"","Parameters.InstanceName == \"TestInstance\"" ]}
```

For "rules", specify a list of string version of rules files. For "data", specify a list of string version of data files.

When `--payload` is specified `--rules` and `--data` cannot be specified.

`-p`, `--print-json`

Prints the output in JSON format.

`-s`, `--show-clause-failures`

Shows clause failure including a summary.

`-V`, `--version`

Prints version information.

`-v`, `--verbose`

Increases the output verbosity. Can be specified multiple times.

`-z`, `--structured`

Prints out a list of structured and valid JSON/YAML. This argument conflicts with the following arguments: verbose, print-json, show-summary: all/fail/pass/skip, output-format: single-line-summary

Options

`-d`, `--data` (string)

Provides a data file or directory of data files in JSON or YAML. Supports passing multiple values by using this option repeatedly.

Example: `--data template1.yaml --data ./data-dir1 --data template2.yaml`

For directory arguments such as `data-dir1` above, scanning is only supported for files with following extensions: .yaml, .yml, .json, .jsn, .template

If you specify the `--payload` flag, don't specify the `--data` option.

`-i`, `--input-parameters` (string)

Provides a parameter file or directory of parameter files in JSON or YAML that specifies any additional parameters to use along with data files to be used as a combined context. All the parameter files passed as input get merged and this combined context is again merged with each file passed as an argument for `data`. Due to this, every file is expected to contain mutually exclusive properties, without any overlap. Supports passing multiple values by using this option repeatedly.

For directory arguments, scanning is only supported for files with following extensions: .yaml, .yml, .json, .jsn, .template

`-o`, `--output-format` (string)

Specifies the format for the output.

Default: `single-line-summary`

Allowed values: `json` | `yaml` | `single-line-summary` | `junit` | `sarif`

`-r`, `--rules` (string)

Provides a rules file or a directory of rules files. Supports passing multiple values by using this option repeatedly.

Example: `--rules rule1.guard --rules ./rules-dir1 --rules rule2.guard`

For directory arguments such as `rules-dir1` above, scanning is only supported for files with following extensions: .guard, .ruleset

If you specify the `--payload` flag, do not specify the `--rules` option.

`--show-summary` (string)

Controls if the summary table needs to be displayed. `--show-summary fail` (default) or `--show-summary pass,fail` (only show rules that did pass/fail) or `--show-summary none` (to turn it off) or `--show-summary all` (to show all the rules that pass, fail or skip).

Default: `fail`

Allowed values: `none` | `all` | `pass` | `fail` | `skip`

`-t`, `--type` (string)

Provides the format of your input data. When you specify the input data type, Guard displays the logical names of CloudFormation template resources in the output. By default, Guard displays property paths and values, such as `Property [/Resources/vol2/Properties/Encrypted`.

Allowed values: `CFNTemplate`

Example

```
cfn-guard validate --data example.json --rules rules.guard
```

Output

If Guard successfully validates the templates, the `validate` command returns an exit status of `0` (`$?` in bash). If Guard identifies a rule violation, the `validate` command returns a status report of the rules that failed.

```
example.json Status = FAIL
FAILED rules
rules.guard/policy_effect_is_deny    FAIL
```

```
---
Evaluation of rules rules.guard against data example.json
--
Property [/path/to/Effect] in data [example.json] is not compliant with
 [policy_effect_is_deny] because provided value ["Allow"] did not match expected value
 ["Deny"]. Error Message [ Policy statement "Effect" must be "Deny".]
```

See also

- Validating input data against Guard rules
- Using input parameters with Guard rules

Security in AWS CloudFormation Guard

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security of the cloud and security in the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Guard, see AWS Services in Scope by Compliance Program.
- **Security in the cloud –** Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations

The following documentation helps you understand how to apply the shared responsibility model when installing Guard as an AWS Lambda function (*cfn-guard-lambda*):

- Security in the *AWS Command Line Interface User Guide*
- Security in the *AWS Lambda Developer Guide*
- Security in the *AWS Identity and Access Management User Guide*

AWS CloudFormation Guard Document history

The following table describes the documentation releases for AWS CloudFormation Guard.

- **Latest documentation update**: July 30, 2025
- **Latest version**: 3.1.2

Change	Description	Date
Documentation update	Updated Guard CLI command reference documentation to align with current implement ation. Version references updated to Guard 3.1.2.	July 30, 2025
Version 3.0.0 release	Version 3.0.0 introduces the following improvements: • Introduction and installat ion topics updated for release of Guard 3.0.0. • Added installation instructi ons for Homebrew and Chocolatey. • Information related to migrating Guard rules updated to reflect changes in Guard version 3.0.0. • Added a prominent link to the AWS CloudFormation Guard GitHub repository.	June 30, 2023
Version 2.1.3 release	Version 2.1.3 introduces the following improvements:	June 9, 2023

	Information on Guard 2.1.3 enhancements has been added. References to Guard 2.0 have been updated to Guard 2.1.3.	
Version 2.0.4 release	Version 2.0.4 introduces the following improvements: The `--payload` flag was added to the `validate` command. For more information, see validate in the Guard CLI reference.	October 19, 2021

Version 2.0.3 release	Version 2.0.3 introduces the following improvements:
	
	• You can provide test names for each test in your unit testing file. For more information, see Testing Guard rules.
	• The following options were added to the `validate` command:
	  • `--output-format`
	  • `--show-summary`
	  • `--type`
	
	For more information, see validate in the Guard CLI reference.
		July 27, 2021

Initial release	Initial release of the AWS CloudFormation Guard User Guide.	July 15, 2021

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.