# policy$_{sentry}$

**Feb 07, 2020**

# Introduction

1 Overview 3

2 Comparison to other tools 9

3 Installation 13

4 Command cheat sheet 15

5 Writing IAM Policies 19

6 Querying the IAM Policy Database 25

7 Docker 29

8 Initialization (Optional) 31

9 Terraform Demo 33

10 Terraform Modules 35

11 Contributing 37

12 Library Usage 47

13 Implementation Strategy 75

14 IAM Policies 77

15 Minimization 81

16 Indices and tables 83

Python Module Index 85

Index 87

Policy Sentry is an AWS IAM Least Privilege Policy Generator, auditor, and analysis database. It compiles database tables based on the AWS IAM Documentation on Actions, Resources, and Condition Keys and leverages that data to create least-privilege IAM policies.

Organizations can use Policy Sentry to:

- **Limit the blast radius in the event of a breach**: If an attacker gains access to user credentials or Instance Profile credentials, access levels and resource access should be limited to the least amount needed to function. This can help avoid situations such as the Capital One breach, where after an SSRF attack, data was accessible from the compromised instance because the role allowed access to all S3 buckets in the account. In this case, Policy Sentry would only allow the role access to the buckets necessary to perform its duties.
- **Scale creation of secure IAM Policies**: Rather than dedicating specialized and talented human resources to manual IAM reviews and creating IAM policies by hand, organizations can leverage Policy Sentry to write the policies for them in an automated fashion.

Policy Sentry's policy writing templates are expressed in YAML and include the following:

- Name and Justification for why the privileges are needed
- CRUD levels (Read/Write/List/Tagging/Permissions management)
- Amazon Resource Names (ARNs), so the resulting policy only points to specific resources and does not grant access to * resources.

Policy Sentry can also be used to:

- Audit IAM Policies based on access levels
- Query the IAM database to reduce manual search time
- Download live policies from an AWS account auditing purposes
- Generate IAM Policies based on Terraform output
- Write least-privilege IAM Policies based on a list of IAM actions (or CRUD levels)

Navigate below to get started with Policy Sentry!

# CHAPTER 1

# Overview

Policy Sentry is an IAM Least Privilege Policy Generator, auditor, and analysis database.

## 1.1 Motivation

Writing security-conscious IAM Policies by hand can be very tedious and inefficient. Many Infrastructure as Code developers have experienced something like this:

- Determined to make your best effort to give users and roles the least amount of privilege you need to perform your duties, you spend way too much time combing through the AWS IAM Documentation on Actions, Resources, and Condition Keys for AWS Services.
- Your team lead encourages you to build security into your IAM Policies for product quality, but eventually you get frustrated due to project deadlines.
- You don't have an embedded security person on your team who can write those IAM policies for you, and there's no automated tool that will automagically sense the AWS API calls that you perform and then write them for you in a least-privilege manner.
- After fantasizing about that level of automation, you realize that writing least privilege IAM Policies, seemingly out of charity, will jeopardize your ability to finish your code in time to meet project deadlines.
- You use Managed Policies (because hey, why not) or you eyeball the names of the API calls and use wildcards instead so you can move on with your life.

Such a process is not ideal for security or for Infrastructure as Code developers. We need to make it easier to write IAM Policies securely and abstract the complexity of writing least-privilege IAM policies. That's why I made this tool.

## 1.2 Authoring Secure IAM Policies

Policy Sentry's flagship feature is that it can create IAM policies based on resource ARNs and access levels. Our CRUD functionality takes the opinionated approach that IAC developers shouldn't have to understand the complexities of AWS IAM - we should abstract the complexity for them. In fact, developers should just be able to say…

- “I need Read/Write/List access to `arn:aws:s3:::example-org-sbx-vmimport`”
- “I need Permissions Management access to `arn:aws:secretsmanager:us-east-1:123456789012:secret:mysec`
- “I need Tagging access to `arn:aws:ssm:us-east-1:123456789012:parameter/test`”

…and our automation should create policies that correspond to those access levels.

How do we accomplish this? Well, Policy Sentry leverages the AWS documentation on Actions, Resources, and Condition Keys documentation to look up the actions, access levels, and resource types, and generates policies according to the ARNs and access levels. Consider the table snippet below:

| Actions | Access Level | Resource Types |
|---|---|---|
| ssm:GetParameter | Read | parameter |
| ssm:DescribeParameters | List | parameter |
| ssm:PutParameter | Write | parameter |
| secretsmanager:PutResourcePolicy | Permissions management | secret |
| secretsmanager:TagResource | Tagging | secret |

Policy Sentry aggregates all of that documentation into a single database and uses that database to generate policies according to actions, resources, and access levels.

To get started, install Policy Sentry:

```
pip3 install --user policy_sentry
```

Then initialize the IAM database:

```
policy_sentry initialize
```

To generate a policy according to resources and access levels, start by creating a template with this command so you can just fill out the ARNs:

```
policy_sentry create-template --name myRole --output-file crud.yml --template-type␣
↪crud
```

It will generate a file like this:

```
mode: crud
name: myRole
description: '' # Insert description
role_arn: '' # Insert the ARN of the role that will use this
read:
- '' # Insert ARNs for Read access
write:
- '' # Insert ARNs...
list:
- '' # Insert ARNs...
tagging:
- '' # Insert ARNs...
permissions-management:
- '' # Insert ARNs...
```

Then just fill it out:

```
mode: crud
name: myRole
description: 'Justification for privileges'
role_arn: 'arn:aws:iam::123456789102:role/myRole'
read:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
write:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
list:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
tagging:
- 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
permissions-management:
- 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
```

Then run this command:

```
policy_sentry write-policy --input-file crud.yml
```

It will generate these results:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SsmReadParameter",
            "Effect": "Allow",
            "Action": [
                "ssm:getparameter",
                "ssm:getparameterhistory",
                "ssm:getparameters",
                "ssm:getparametersbypath",
                "ssm:listtagsforresource"
            ],
            "Resource": [
                "arn:aws:ssm:us-east-1:123456789012:parameter/myparameter"
            ]
        },
        {
            "Sid": "SsmWriteParameter",
            "Effect": "Allow",
            "Action": [
                "ssm:deleteparameter",
                "ssm:deleteparameters",
                "ssm:putparameter",
                "ssm:labelparameterversion"
            ],
            "Resource": [
                "arn:aws:ssm:us-east-1:123456789012:parameter/myparameter"
            ]
        },
        {
            "Sid": "SecretsmanagerPermissionsmanagementSecret",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:deleteresourcepolicy",
                "secretsmanager:putresourcepolicy"
```

(continues on next page)

(continued from previous page)

```
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
            ]
        },
        {
            "Sid": "SecretsmanagerTaggingSecret",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:tagresource",
                "secretsmanager:untagresource"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
            ]
        }
    ]
}
```

Notice how the policy above recognizes the ARNs that the user supplies, along with the requested access level. For instance, the SID "SecretsmanagerTaggingSecret" contains Tagging actions that are assigned to the secret resource type only.

This rapidly speeds up the time to develop IAM policies, and ensures that all policies created limit access to exactly what your role needs access to. This way, developers only have to determine the resources that they need to access, and we abstract the complexity of IAM policies away from their development processes.

## 1.3 Installation

- Policy Sentry is available via pip. To install, run:

```
pip3 install --user policy_sentry
```

### 1.3.1 Shell completion

To enable Bash completion, put this in your *.bashrc*:

```
eval "$(_POLICY_SENTRY_COMPLETE=source policy_sentry)"
```

To enable ZSH completion, put this in your *.zshrc*:

```
eval "$(_POLICY_SENTRY_COMPLETE=source_zsh policy_sentry)"
```

## 1.4 Usage

- `create-template`: Creates the YML file templates for use in the `write-policy` command types.
- `write-policy`: Leverage a YAML file to write policies for you
    - Option 1: Specify CRUD levels (Read, Write, List, Tagging, or Permissions management) and the ARN of the resource. It will write this for you. See the documentation on CRUD mode

  - Option 2: Specify a list of actions. It will write the IAM Policy for you, but you will have to fill in the ARNs. See the documentation on Action Mode.
- `write-policy-dir`: This can be helpful in the Terraform use case. For more information, see the wiki.
- `query`: Query the IAM database tables. This can help when filling out the Policy Sentry templates, or just querying the database for quick knowledge. - Option 1: Query the Actions Table (`action-table`) - Option 2: Query the ARNs Table (`arn-table`) - Option 3: Query the Conditions Table (`condition-table`)
- `initialize`: (Optional) Create a SQLite database that contains all of the services available through the Actions, Resources, and Condition Keys documentation. See the documentation.

## 1.5 Author Information

Author:

- Kinnaird McQuade
  - Twitter
  - Keybase
  - LinkedIn

Contributors:

- Matt Jones
  - Twitter
  - Keybase
  - LinkedIn

# CHAPTER 2

# Comparison to other tools

## 2.1 Policy Revocation Tools

### 2.1.1 Repokid

RepoKid is a popular tool that was developed by Netflix, and is one of the more mature and battle-tested AWS IAM open source projects. It leverages AWS Access Advisor, which informs you how many AWS services your IAM Principal has access to, and how many of those services it has used in the last X amount of days or months. If you haven't used a service within the last 30 days, it "repos" your policy, and strips it of the privileges it doesn't use. It has some advanced features to allow for whitelisting roles and overall is a great tool.

One shortcoming is that AWS IAM Access Advisor only provides details at the service level (ex: S3-wide, or EC2-wide) and not down to the IAM Action level, so the revised policy is not very granular. However, RepoKid plays a unique role in the IAM ecosystem right now in that there are not any open source tools that provide similar functionality. **For that reason, it is best to view RepoKid and Policy Sentry as complimentary.**

Travis McPeak summarized the potential dynamic between Policy Sentry and RepoKid very well on Clint Gliber's blog:

> Policy Sentry aims to make it easy to create initial least privilege policies and then Repokid takes away unused permissions.
>
> Creating policies is difficult, so Policy Sentry creates policies based on top level goals and target resources, and then on the backend substitutes the applicable action list to generate the policy. This is very helpful for anybody creating the first version of a policy.
>
> To help with simplicity these permissions will be assigned somewhat coarsely. So Repokid can use data to remove the specific actions that were granted and aren't required. Also Repokid will repo down unused permissions once an application stops being used or scope changes.

## 2.2 AWS Tools

### 2.2.1 AWS Console - Visual Policy Editor

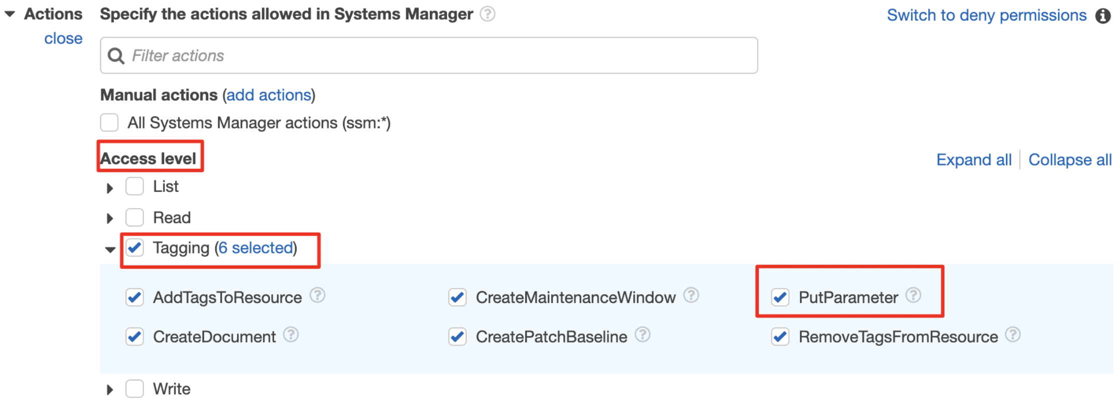
- AWS IAM Visual Policy Editor in the AWS Console

This policy generator is great in general and you should use it if you're getting used to writing IAM policies.

It's very similar to `policy_sentry` - you are able to bulk select according to access levels.

However, there are a number of downsides:

- **Missing access level type**: It does not specifically flag "Permissions management" access level
- **No override capabilities for inaccurate Access Levels**: Note how the `ssm:PutParameter` action is listed as "Tagging". This is inaccurate; it should be "Write". Policy_sentry allows you to override inaccurate access levels, whereas the Visual Policy Editor has had inaccurate Access levels for the last several years without any fixes.



Specify the actions allowed in Systems Manager

Manual actions (add actions)

☐ All Systems Manager actions (ssm:*)

Access level

☐ List
☐ Read
☑ Tagging (6 selected)
  ☑ AddTagsToResource
  ☑ CreateMaintenanceWindow
  ☑ PutParameter
  ☑ CreateDocument
  ☑ CreatePatchBaseline
  ☑ RemoveTagsFromResource
☐ Write

- **Not automated**: Policy Sentry is, by design, meant for automated policy generation, whereas the Visual Policy Editor is meant to be manual.
- **Console Access**: It also requires access to the AWS Console.
- **Extensibility**: It's open source and Pull Requests are welcome! With `policy_sentry`, we get more control.

On the positive side, it does walk you through creating policies with IAM Condition keys. However, we believe that policy_sentry's approach, where we **always** have policies restricted to the least amount of resources - provides a greater benefit to the end user. Furthermore, we plan on supporting condition keys at some point in the future.

### 2.2.2 AWS Policy Generator (static website)

- AWS Policy Generator - static website

AWS Policy Generator is a great tool; it supports IAM policies, as well as multiple types of resource-based policies (SQS Queue policy, S3 bucket policy, SNS Topic Policy, and VPC Endpoint Policy).

**Loose ARN formatting**: The regex expressions that it uses per-service does not require that actual valid resource ARNs are met - just that they meet the Regex requirement, which is uniform per-service. It just isn't as accurate or up to date as the actual IAM policy generation through the AWS Console

**Missing actions**: To determine the list of actions, it relies on a file titled policies.js, which contains a list of IAM Actions. However, this file is not as well maintained as the Actions, Resources, and Condition Keys tables. For example, it does not have these actions:

```
a4b:describe*
appstream:get*
cloudformation:preview*
codestar:verify*
ds:check*
health:get*
health:list*
kinesisanalytics:get*
lightsail:list*
mobilehub:validate*
resource-groups:describe*
```

## 2.3 Log-based Policy Generators

### 2.3.1 CloudTracker

- CloudTracker

Policy Sentry is somewhat similar to CloudTracker. CloudTracker queries CloudTrail logs using Amazon Athena and attempts to "guess" the matching between CloudTrail actions and IAM actions, then generates a policy. Given that there is not a 1-to-1 mapping between the names of Actions listed in CloudTrail log entries and the names AWS IAM Actions, the results are not always accurate. It is a good place to start, but the generated policies all contain Resources: "*", so it is up to the user to restrict those IAM actions to only the necessary resources.

### 2.3.2 Trailscraper

- Trailscraper

Trailscraper does automated policy generation from CloudTrail logs, but there are some major limitations:

1. The generated policies have Resources set to *', not to a specific resource ARN
2. It downloads all of the CloudTrail logs. This takes a while.
   - Cloudtracker (https://github.com/duo-labs/cloudtracker) uses Amazon Athena, which is more efficient. In the future, I'd like to see a combined approach between all three of these tools to generate IAM policies based on Cloudtrail logs. 3. It is accurate to the point where there is a 1-to-1 mapping with the IAM actions vs CloudTrail logs. As I mentioned in other comments, since not every IAM Action is logged in CloudTrail and not every CloudTrail action matches IAM Actions, the results are not always accurate.

## 2.4 Other Infrastructure as Code Tools

### 2.4.1 aws-iam-generator

- aws-iam-generator

aws-iam-generator still requires you to write the actual policy templates from scratch, and then they allow you to re-use those policy templates.

Consider the JSON under this area of their README.

It's essentially a method for managing their policies as code - but it doesn't make those policies restricted to certain resources, unless you configure it that way. Using `policy_sentry --write-policy`, you have to supply a file with resource ARNs, and it will write the policy for you, rather than supplying a policy file, and hoping the ARNs fit that use case.

### 2.4.2 Terraform

The rationale described above also generally applies to Terraform, in that it still requires you to write the actual policy templates from scratch, and then you can re-use those policy templates. However, you still need to make those policies secure by default.

CHAPTER 3

# Installation

- `policy_sentry` is available via pip (Python 3 only). To install, run:

```
pip3 install --user policy_sentry
```

## 3.1 Shell completion

To enable Bash completion, put this in your *.bashrc*:

```
eval "$(_POLICY_SENTRY_COMPLETE=source policy_sentry)"
```

To enable ZSH completion, put this in your *.zshrc*:

```
eval "$(_POLICY_SENTRY_COMPLETE=source_zsh policy_sentry)"
```

# CHAPTER 4

# Command cheat sheet

## 4.1 Commands

- `create-template`: Creates the YML file templates for use in the `write-policy` command types.
- `write-policy`: Leverage a YAML file to write policies for you
    - Option 1: CRUD Mode. Specify CRUD levels (Read, Write, List, Tagging, or Permissions management) and the ARN of the resource. It will write this for you. See the documentation for more details.
    - Option 2: Actions Mode. Specify a list of actions. It will write the IAM Policy for you, but you will have to fill in the ARNs. See the documentation for more details.
- `write-policy-dir`: This can be helpful in the Terraform use case. For more information, see the wiki.
- `query`: Query the IAM database tables. This can help when filling out the Policy Sentry templates, or just querying the database for quick knowledge.
    - Option 1: Query the Actions Table (`--table action`)
    - Option 2: Query the ARNs Table (`--table arn`)
    - Option 3: Query the Conditions Table (`--table condition`)
- `initialize`: (Optional) Create a SQLite database that contains all of the services available through the Actions, Resources, and Condition Keys documentation. See the documentation.

## 4.2 Policy Writing Commands

```
# Create templates first!!! This way you can just paste the values you need rather␣
↪than remembering the YAML format
# CRUD mode
policy_sentry create-template --name myRole --output-file tmp.yml --template-type crud
# Actions mode
policy_sentry create-template --name myRole --output-file tmp.yml --template-type␣
↪actions
```

(continues on next page)

(continued from previous page)

```
# Get a list of actions that do not support resource constraints
policy_sentry query action-table --service s3 --wildcard-only --fmt yaml

# Get a list of actions at the "Write" level in S3 that do not support resource␣
↪constraints
policy_sentry query action-table --service s3 --access-level write --wildcard-only --
↪fmt yaml

# Initialize the policy_sentry config folder and create the IAM database tables.
policy_sentry initialize

# Write policy based on resource-specific access levels
policy_sentry write-policy --input-file examples/yml/crud.yml

# Write policy_sentry YML files based on resource-specific access levels on a␣
↪directory basis
policy_sentry write-policy-dir --input-dir examples/input-dir --output-dir examples/
↪output-dir

# Write policy based on a list of actions
policy_sentry write-policy --input-file examples/yml/actions.yml
```

## 4.3 IAM Database Query Commands

- Query the **Action** table:

```
# Get a list of all IAM actions across ALL services that have "Permissions management
↪" access
policy_sentry query action-table --service all --access-level permissions-management

# Get a list of all IAM Actions available to the RAM service
policy_sentry query action-table --service ram

# Get details about the `ram:TagResource` IAM Action
policy_sentry query action-table --service ram --name tagresource

# Get a list of all IAM actions under the RAM service that have the Permissions␣
↪management access level.
policy_sentry query action-table --service ram --access-level permissions-management

# Get a list of all IAM actions under the SES service that support the␣
↪`ses:FeedbackAddress` condition key.
policy_sentry query action-table --service ses --condition ses:FeedbackAddress
```

- Query the **ARN** table:

```
# Get a list of all RAW ARN formats available through the SSM service.
policy_sentry query arn-table --service ssm

# Get the raw ARN format for the `cloud9` ARN with the short name `environment`
policy_sentry query arn-table --service cloud9 --name environment
```

(continues on next page)

(continued from previous page)

```
# Get key/value pairs of all RAW ARN formats plus their short names
policy_sentry query arn-table --service cloud9 --list-arn-types
```

- Query the **Condition Keys** table:

```
# Get a list of all condition keys available to the Cloud9 service
policy_sentry query condition-table --service cloud9
# Get details on the condition key titled `cloud9:Permissions`
policy_sentry query condition-table --service cloud9 --name cloud9:Permissions
```

## 4.4 Initialization (Optional)

```
# Initialize the policy_sentry config folder and create the IAM database tables.
policy_sentry initialize

# Fetch the most recent version of the AWS documentation so you can experiment with␣
↪new services.
policy_sentry initialize --fetch

# Override the Access Levels by specifying your own Access Levels (example:,␣
↪correcting Permissions management levels)
policy_sentry initialize --access-level-overrides-file ~/.policy_sentry/access-level-
↪overrides.yml
policy_sentry initialize --access-level-overrides-file ~/.policy_sentry/overrides-
↪resource-policies.yml
```

CHAPTER 5

# Writing IAM Policies

## 5.1 CRUD Mode

- **TLDR**: Building IAM policies with resource constraints and access levels.

This is the flagship feature of this tool. You can just specify the CRUD levels (Read, Write, List, Tagging, or Permissions management) for each action in a YAML File. The policy will be generated for you. You might need to fiddle with the results for your use in Terraform, but it significantly reduces the level of effort to build least privilege into your policies.

### 5.1.1 Command options

- `--input-file`: YAML file containing the CRUD levels + Resource ARNs. Required.
- `--minimize`: Whether or not to minimize the resulting statement with *safe* usage of wildcards to reduce policy length. Set this to the character length you want. This can be extended for readability. I suggest setting it to `0`.
- `--quiet`: Set the logging level to WARNING instead of INFO.

Example:

```
policy_sentry write-policy --input-file examples/crud.yml
```

### 5.1.2 Instructions

- To generate a policy according to resources and access levels, start by creating a template with this command so you can just fill out the ARNs:

```
policy_sentry create-template --name myRole --output-file crud.yml --template-type␣
↪crud
```

- It will generate a file like this:

```
mode: crud
name: myRole
description: ''
role_arn: ''
# Insert ARNs below
read:
- ''
write:
- ''
list:
- ''
tagging:
- ''
permissions-management:
- ''
# Provide a list of IAM actions that cannot be restricted to ARNs
wildcard:
- ''
```

- Then just fill it out:

```
mode: crud
name: myRole
description: 'Justification for privileges'
role_arn: 'arn:aws:iam::123456789102:role/myRole'
read:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
write:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
list:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
tagging:
- 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
permissions-management:
- 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
```

- Run the command:

```
policy_sentry write-policy --input-file examples/crud.yml
```

- It will generate an IAM Policy containing an IAM policy with the actions restricted to the ARNs specified above.
- The resulting policy (without the `--minimize command`) will look like this:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SsmReadParameter",
            "Effect": "Allow",
            "Action": [
                "ssm:getparameter",
                "ssm:getparameterhistory",
                "ssm:getparameters",
                "ssm:getparametersbypath",
                "ssm:listtagsforresource"
            ],
            "Resource": [
```

(continues on next page)

(continued from previous page)

```
                "arn:aws:ssm:us-east-1:123456789012:parameter/myparameter"
            ]
        },
        {
            "Sid": "SsmWriteParameter",
            "Effect": "Allow",
            "Action": [
                "ssm:deleteparameter",
                "ssm:deleteparameters",
                "ssm:putparameter",
                "ssm:labelparameterversion"
            ],
            "Resource": [
                "arn:aws:ssm:us-east-1:123456789012:parameter/myparameter"
            ]
        },
        {
            "Sid": "SecretsmanagerPermissionsmanagementSecret",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:deleteresourcepolicy",
                "secretsmanager:putresourcepolicy"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
            ]
        },
        {
            "Sid": "SecretsmanagerTaggingSecret",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:tagresource",
                "secretsmanager:untagresource"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
            ]
        }
    ]
}
```

## 5.2 Actions Mode

- **TLDR**: Supply a list of actions in a YAML file and generate the policy accordingly.

### 5.2.1 Command options

- `--input-file`: YAML file containing the list of actions
- `--minimize`: Whether or not to minimize the resulting statement with *safe* usage of wildcards to reduce policy length. Set this to the character lengh you want - for example, 4
- `--quiet`: Set the logging level to WARNING instead of INFO.

Example:

```
policy_sentry write-policy --input-file examples/actions.yml
```

### 5.2.2 Instructions

- If you already know the IAM actions, you can just run this command to create a template to fill out:

```
policy_sentry create-template --name myRole --output-file actions.yml --template-type␣
↪actions
```

- It will generate a file with contents like this:

```
mode: actions
name: myRole
description: '' # Insert value here
role_arn: '' # Insert value here
actions:
- ''  # Fill in your IAM actions here
```

- Create a yaml file with the following contents:

```
mode: actions
name: 'RoleNameWithActions'
description: 'Justification for privileges' # for auditability
role_arn: 'arn:aws:iam::123456789102:role/myRole' # for auditability
actions:
- kms:CreateGrant
- kms:CreateCustomKeyStore
- ec2:AuthorizeSecurityGroupEgress
- ec2:AuthorizeSecurityGroupIngress
```

- Then run this command:

```
policy_sentry write-policy --input-file actions.yml
```

- The output will look like this:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "KmsPermissionsmanagementKey",
            "Effect": "Allow",
            "Action": [
                "kms:creategrant"
            ],
            "Resource": [
                "arn:aws:kms:${Region}:${Account}:key/${KeyId}"
            ]
        },
        {
            "Sid": "Ec2WriteSecuritygroup",
            "Effect": "Allow",
            "Action": [
                "ec2:authorizesecuritygroupegress",
                "ec2:authorizesecuritygroupingress"
```

(continues on next page)

(continued from previous page)

```
            ],
            "Resource": [
                "arn:aws:ec2:${Region}:${Account}:security-group/${SecurityGroupId}"
            ]
        },
        {
            "Sid": "MultMultNone",
            "Effect": "Allow",
            "Action": [
                "kms:createcustomkeystore",
                "cloudhsm:describeclusters"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

## 5.3 Folder Mode

- **TLDR**: Write Multiple Policies from CRUD mode templates

This command provides the same function as *write-policy*'s CRUD mode, but it can execute all the CRUD mode files in a folder. This is particularly useful in the Terraform use case, where the Terraform module can export a number of Policy Sentry template files into a folder, which can then be consumed using this command.

See the Terraform demo for more details.

```
Usage: policy_sentry write-policy-dir [OPTIONS]

Options:
  --input-dir TEXT    Relative path to Input directory that contains policy_sentry .
↪yml files (CRUD mode only)  [required]
  --output-dir TEXT   Relative path to directory to store AWS JSON policies [required]
  --minimize INTEGER  Minimize the resulting statement with *safe* usage of wildcards␣
↪to reduce policy length. Set this to the character length you want - for example, 4
  --quiet             Set the logging level to WARNING instead of INFO.
  --help              Show this message and exit.
```

# CHAPTER 6

# Querying the IAM Policy Database

Policy Sentry relies on a SQLite database, generated at *initialize* time, which contains all of the services available through the Actions, Resources, and Condition Keys documentation. The HTML files from that AWS documentation is scraped and stored in the SQLite database, which is then stored in `$HOME/.policy_sentry/aws.sqlite3`.

Policy Sentry supports querying that database through the CLI. This can help with writing policies and generally knowing what values to supply in your policies.

## 6.1 Commands

- Query the **Action** table:

```
# NOTE: Use --fmt yaml or --fmt json to change the output format. Defaults to json␣
↪for querying

# Get a list of actions that do not support resource constraints
policy_sentry query action-table --service s3 --wildcard-only --fmt yaml

# Get a list of actions at the "Read" level in S3 that do not support resource␣
↪constraints
policy_sentry query action-table --service s3 --access-level read --wildcard-only --
↪fmt yaml

# Get a list of all IAM Actions available to the RAM service
policy_sentry query action-table --service ram

# Get details about the `ram:TagResource` IAM Action
policy_sentry query action-table --service ram --name tagresource

# Get a list of all IAM actions under the RAM service that have the Permissions␣
↪management access level.
policy_sentry query action-table --service ram --access-level permissions-management
```

(continues on next page)

(continued from previous page)

```
# Get a list of all IAM actions under the SES service that support the␣
↪`ses:FeedbackAddress` condition key.
policy_sentry query action-table --service ses --condition ses:FeedbackAddress
```

- Query the **ARN** table:

```
# Get a list of all RAW ARN formats available through the SSM service.
policy_sentry query arn-table --service ssm

# Get the raw ARN format for the `cloud9` ARN with the short name `environment`
policy_sentry query arn-table --service cloud9 --name environment

# Get key/value pairs of all RAW ARN formats plus their short names
policy_sentry query arn-table --service cloud9 --list-arn-types
```

- Query the **Condition Keys** table:

```
# Get a list of all condition keys available to the Cloud9 service
policy_sentry query condition-table --service cloud9

# Get details on the condition key titled `cloud9:Permissions`
policy_sentry query condition-table --service cloud9 --name cloud9:Permissions
```

## 6.2 Options

- action-table

```
Usage: policy_sentry query action-table [OPTIONS]

Options:
  --service TEXT                  Filter according to AWS service.  [required]
  --name TEXT                     The name of IAM Action. For example, if the
                                  action is "iam:ListUsers", supply
                                  "ListUsers" here.
  --access-level [read|write|list|tagging|permissions-management]
                                  If action table is chosen, you can use this
                                  to filter according to CRUD levels.
                                  Acceptable values are read, write, list,
                                  tagging, permissions-management
  --condition TEXT                If action table is chosen, you can supply a
                                  condition key to show a list of all IAM
                                  actions that support the condition key.
  --wildcard-only                 If action table is chosen, show the IAM
                                  actions that only support wildcard resources
                                  - i.e., cannot support ARNs in the resource
                                  block.
  --fmt [yaml|json]               Format output as YAML or JSON. Defaults to
                                  "yaml"
  --quiet             Set the logging level to WARNING instead of INFO.
  --help                          Show this message and exit.
```

- arn-table

```
Usage: policy_sentry query arn-table [OPTIONS]

  Query the ARN Table from the Policy Sentry database

Options:
  --service TEXT     Filter according to AWS service.  [required]
  --name TEXT        The short name of the resource ARN type. For example,
                     `bucket` under service `s3`.
  --list-arn-types   Show the short names of ARN Types. If empty, this will
                     show RAW ARNs only.
  --fmt [yaml|json]  Format output as YAML or JSON. Defaults to "yaml"
  --quiet             Set the logging level to WARNING instead of INFO.
  --help             Show this message and exit.
```

- condition-table

```
Usage: policy_sentry query condition-table [OPTIONS]

  Query the condition keys table from the Policy Sentry database

Options:
  --name TEXT        Get details on a specific condition key. Leave this blank
                     to get a list of all condition keys available to the
                     service.
  --service TEXT     Filter according to AWS service.  [required]
  --fmt [yaml|json]  Format output as YAML or JSON. Defaults to "yaml"
  --quiet             Set the logging level to WARNING instead of INFO.
  --help             Show this message and exit.
```

# CHAPTER 7

# Docker

If you prefer using Docker instead of installing the script with Python, we support that as well.

Use this to build the docker image:

```
docker build -t kmcquade/policy_sentry .
```

Use this to run some basic commands:

```
# Basic commands with no arguments
docker run -i --rm kmcquade/policy_sentry:latest "--help"
docker run -i --rm kmcquade/policy_sentry:latest "query"

# Query the database
docker run -i --rm kmcquade/policy_sentry:latest "query action-table --service all --
↪access-level permissions-management"
```

The *write-policy* command also supports passing in the YML config via STDIN. Try it out here:

```
# Write policies by passing in the config via STDIN
cat examples/yml/crud.yml | docker run -i --rm kmcquade/policy_sentry:latest "write-
↪policy"
cat examples/yml/actions.yml | docker run -i --rm kmcquade/policy_sentry:latest
↪"write-policy"
```

## CHAPTER 8

# Initialization (Optional)

*initialize*: This will create a SQLite database that contains all of the services available through the Actions, Resources, and Condition Keys documentation.

Note: This step is now optional. Typical use cases for running the initialize command are: * If you want to run *–fetch* and build the latest database from the AWS Docs. This is good if you want to try out the latest cool services. * If you want to verify the database contents on your own. * If you want to build the SQLite database from the raw HTML files, rather than copying it from the package.

The database is stored in `$HOME/.policy_sentry/aws.sqlite3`.

The database is generated based on the HTML files stored in the `policy_sentry/shared/data/docs/` directory.

## 8.1 Options

- `--access-level-overrides-file` (Optional): Path to your own custom access level overrides file, used to override the Access Levels per action provided by AWS docs. The default one is here.
- `--fetch` (Optional): Specify this flag to fetch the HTML Docs directly from the AWS website. This will be helpful if the docs in the Git repository are behind the live docs and you need to use the latest version of the docs right now.
- `--build` (Optional) Build the SQLite database from the HTML files rather than copying the SQLite database file from the python package. Defaults to false.

## 8.2 Usage

```
# Initialize the database, using the existing Access Level Overrides file
policy_sentry initialize

# Fetch the most recent version of the AWS documentation so you can experiment with␣
↪new services.
```

(continues on next page)

(continued from previous page)

```
# This can be helpful in case the AWS HTML files in the Python package are outdated,␣
↪even if it is a week old
policy_sentry initialize --fetch

# Build the database file from the HTML files rather than using the bundled binary.
policy_sentry initialize --build

# Initialize the database with a custom Access Level Overrides file

policy_sentry initialize --access-level-overrides-file ~/.policy_sentry/access-level-
↪overrides.yml
policy_sentry initialize --access-level-overrides-file ~/.policy_sentry/overrides-
↪resource-policies.yml
```

## 8.3 Skipping Initialization

When using Policy Sentry manually, you have to build a local database file with the initialize function.

However, if you are developing your own Python code and you want to import Policy Sentry as a third party package, you can skip the initialization and leverage the local database file that is bundled with the Python package itself.

This is especially useful for developers who wish to leverage Policy Sentry's capabilities that require the use of the IAM database (such as querying the IAM database table). This way, you don't have to initialize the database and can just query it immediately.

## CHAPTER 9

# Terraform Demo

Please download the demo code here to follow along.

## 9.1 Command options

```
Usage: policy_sentry write-policy-dir [OPTIONS]

  write_policy, but this time with an input directory of YML/YAML files, and
  an output directory for all the JSON files

Options:
  --input-dir TEXT    Relative path to Input directory that contains policy_sentry .
↪yml files (CRUD mode only)  [required]
  --output-dir TEXT   Relative path to directory to store AWS JSON policies [required]
  --minimize INTEGER  Minimize the resulting statement with *safe* usage of wildcards␣
↪to reduce policy length. Set this to the character length you want - for example, 4
  --help              Show this message and exit.
```

## 9.2 Prerequisites

This requires:

- Terraform v0.12.8
- AWS credentials; must be authenticated

## 9.3 Tutorial

- Install policy_sentry

```
pip3 install policy_sentry
```

- Initialize policy_sentry

```
policy_sentry initialize
```

- Execute the first Terraform module:

```
cd environments/standard-resources
tfjson install 0.12.8
terraform init
terraform plan
terraform apply -auto-approve
```

This will create a YML file to be used by policy_sentry in the environments/iam-resources/files/ directory titled example-role-randomid.yml.

- Write the policy using policy_sentry:

```
cd ../iam-resources
policy_sentry write-policy-dir --input-dir files --output-dir files
```

This will create a JSON file to be consumed by Terraform's `aws_iam_policy` resource to create an IAM policy.

- Now create the policies with Terraform:

```
terraform init
terraform plan
terraform apply -auto-approve
```

- Don't forget to cleanup

```
terraform destroy -auto-approve
cd ../standard-resources
terraform destroy -auto-approve
```

CHAPTER 10

# Terraform Modules

## 10.1 1: Install policy_sentry

- Install policy_sentry

```
pip3 install policy_sentry
```

- Initialize policy_sentry

```
policy_sentry initialize
```

## 10.2 2: Generate the policy_sentry YAML File

Create a file with the following in `some-directory`:

```
module "policy_sentry_yml" {
  source           = "git::https://github.com/salesforce/policy_sentry.git//examples/
↪terraform/modules/generate-policy_sentry-yml"
  role_name        = ""
  role_description = ""
  role_arn         = ""

  list_access_level                   = []
  permissions_management_access_level = []
  read_access_level                   = []
  tagging_access_level                = []
  write_access_level                  = []

  yml_file_destination_path = "../other-directory/files"
}
```

Make sure you fill out the actual directory path properly. Note that `yml_file_destination_path` should point to the directory mentioned in Step 3.

## 10.3 3: Run policy_sentry and specify proper target directory

- Enter the directory you specified under `yml_file_destination_path` above.
- Run the following:

```
policy_sentry write-policy-dir --input-dir files --output-dir files
```

## 10.4 4: Create the IAM Policies using JSON files from directory

Then from `other-directory`:

```
module "policies" {
  source = "git::https://github.com/salesforce/policy_sentry.git//examples/terraform/
↪modules/generate-iam-policies"
  relative_path_to_json_policy_files = "files"
}
```

# CHAPTER 11

# Contributing

Want to contribute back to Policy Sentry? This page describes the general development flow, our philosophy, the test suite, and issue tracking.

**Impostor Syndrome Disclaimer**

Before we get into the details: *We want your help. No, really.*

There may be a little voice inside your head that is telling you that you're not ready to be an open source contributor; that your skills aren't nearly good enough to contribute. What could you possibly offer a project like this one?

We assure you – the little voice in your head is wrong. If you can write code at all, you can contribute code to open source. Contributing to open source projects is a fantastic way to advance one's coding skills. Writing perfect code isn't the measure of a good developer (that would disqualify all of us!); it's trying to create something, making mistakes, and learning from those mistakes. That's how we all improve.

We've provided some clear Contribution Guidelines that you can read here. The guidelines outline the process that you'll need to follow to get a patch merged. By making expectations and process explicit, we hope it will make it easier for you to contribute.

And you don't just have to write code. You can help out by writing documentation, tests, or even by giving feedback about this work. (And yes, that includes giving feedback about the contribution guidelines.)

(Adrienne Friend came up with this disclaimer language.)

## 11.1 Contributing to Documentation

If you're looking to help document Policy Sentry, your first step is to get set up with Sphinx, our documentation tool. First you will want to make sure you have a few things on your local system:

- python-dev (if you're on OS X, you already have this)
- pip
- pipenv

Once you've got all that, the rest is simple:

```
# If you have a fork, you'll want to clone it instead
git clone git@github.com:salesforce/policy_sentry.git

# Set up the Pipenv
pipenv install --skip-lock
pipenv shell

# Enter the docs directory and compile
cd docs/
make html

# View the file titled docs/_build/html/index.html in your browser
```

### 11.1.1 Building Documentation

Inside the `docs` directory, you can run `make` to build the documentation. See `make help` for available options and the Sphinx Documentation for more information.

### 11.1.2 Docstrings

The comments under each Python Module are Docstrings. We use those to generate our documentation. See more information here: https://sphinx-rtd-tutorial.readthedocs.io/en/latest/build-the-docs.html#generating-documentation-from-docstrings.

Use the Google style for Docstrings, as shown here: http://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html#google-vs-numpy

**::**

> **def func(arg1, arg2):** “”“Summary line.
>
> > Extended description of function.
> >
> > **Args:** arg1 (int): Description of arg1 arg2 (str): Description of arg2
> >
> > **Returns:** bool: Description of return value
> >
> > “”” return True

## 11.2 IAM Database

Policy Sentry leverages HTML files from the Actions, Resources, and Condition Keys page AWS documentation to build the IAM database.

- These HTML files are included as part of the PyPi package
- The database itself is

This design choice was made for a few reasons:

1. **Don’t break because of AWS**: The automation must **not** break if the AWS website is down, or if AWS drastically changes the documentation.
2. **Replicability**: Two `git clones` that build the SQLite database should always have the same results
3. **Easy to review**: The repository itself should contain easy-to-understand and easy-to-view documentation, which the user can replicate, to verify with the human eye that no malicious changes have been made.

  - This means no JSON files with complicated structures, or Binary files (the latter of which does not permit `git diff`) in the repository.
  - This helps to mitigate the concern that open source software could be modified to alter IAM permissions at other organizations.

### 11.2.1 How Policy Sentry uses the IAM database

policy_sentry follows this process for generating policies.

1. If the **User-supplied actions** template is provided:
   - Look up the actions in our master Actions Table in the database, which contains the Action Tables for all AWS services
   - If the action in the database matches the actions requested by the user, determine the ARN Format required.
   - Proceed to step 3
2. If **User-supplied ARNs with Access levels** template was provided:
   - Match the user-supplied ARNs with ARN formats in our ARN Table database, which contains the ARN tables for all AWS Services
   - If it matches, get the access level requested by the user
   - Proceed to step 3
3. Compile those into groups, sorted by an SID namespace. The SID namespace follows the format of **Service**, **Access Level**, and **Resource ARN Type**, with no character delimiter (to meet AWS IAM Policy formatting expectations). For example, the namespace could be `SsmReadParameter`, `KmsReadKey`, or `Ec2TagInstance`.
4. Then, we associate the user-supplied ARNs matching that namespace with the SID.
5. If **User-supplied actions** template was provided:
   - Associate the IAM actions requested by the user to the service, access level, and ARN type matching the aforementioned SID namespace
6. If the **User-supplied ARNs with Access levels** template was provided:
   - Associate all the IAM actions that correspond to the service, access level, and ARN type matching that SID namespace.
7. Print the policy

#### Updating the AWS HTML files

The command shown below downloads the Actions, Resources, and Condition Keys pages per-service to the `policy_sentry/shared/data/docs` folder.

- The HTML files will be stored in *policy_sentry/shared/data/docs/list_*.partial.html*
- It also add a file titled `policy_sentry/shared/data/links.yml` as well.
- It also builds a SQLite database file to include as part of the PyPi package.

This will update the

```
python3 ./utils/download_docs.py
```

This downloads the Actions, Resources, and Condition Keys pages per-service to the `policy_sentry/shared/data/docs` folder. It also add a file titled `policy_sentry/shared/data/links.yml` as well.

When a user runs `policy_sentry initialize`, these files are copied over to the config folder (`~/.policy_sentry/`).

This design choice was made for a few reasons:

1. **Don't break because of AWS**: The automation must **not** break if the AWS website is down, or if AWS drastically changes the documentation.
2. **Replicability**: Two *git clones* that build the SQLite database should always have the same results
3. **Easy to review: The repository itself should contain easy-to-understand and easy-to-view documentation, which the user c**

    - This means no JSON files with complicated structures, or Binary files (the latter of which does not permit `git diff`) in the repository.
    - This helps to mitigate the concern that open source software could be modified to alter IAM permissions at other organizations.

## 11.3 Testing

### 11.3.1 Pipenv

```
pipenv --python 3.7  # create the environment
pipenv shell         # start the environment
pipenv install       # install both development and production dependencies
```

### 11.3.2 Invoke

To run and develop Policy Sentry without having to install from PyPi, you can use Invoke.

```
# List available tasks
invoke -l

# that will show the following options:
Available tasks:

  build.build-package           Build the policy_sentry package from the current
                                directory contents for use with PyPi
  build.install-package         Install the policy_sentry package built from the
                                current directory contents (not PyPi)
  build.uninstall-package       Uninstall the policy_sentry package
  build.upload-prod             Upload the package to the PyPi production server
                                (requires credentials)
  build.upload-test             Upload the package to the TestPyPi server
                                (requires credentials)
  docs.make-html                Make the HTML docs locally
  docs.open-html-docs           Open HTML docs in Google Chrome locally on your
                                computer
  docs.remove-html-files        Remove the html files
  integration.analyze-policy    Integration testing: Tests the `analyze`
                                functionality
```

(continues on next page)

(continued from previous page)

```
  integration.clean             Runs `rm -rf $HOME/.policy_sentry`
  integration.initialize        Integration testing: Initialize the
                                policy_sentry database
  integration.query             Integration testing: Tests the `query`
                                functionality (querying the IAM database)
  integration.query-yaml        Integration testing: Tests the `query`
                                functionality (querying the IAM database) - but
                                with yaml
  integration.version           Print the version
  integration.write-policy      Integration testing: Tests the `write-policy`
                                function.
  test.lint                     Linting with `pylint` and `autopep8`
  test.security                 Runs `bandit` and `safety check`
  unit.nose                     Unit testing: Runs unit tests using `nosetests`
  unit.pytest                   Unit testing: Runs unit tests using `pytest`

# To run them, specify `invoke` plus the options:
invoke build.build-package

invoke integration.clean
invoke integration.initialize
invoke integration.analyze-policy
invoke integration.query
invoke integration.write-policy

invoke test.lint
invoke test.security

invoke unit.nose
```

### 11.3.3 Local Unit Testing and Integration Testing: Quick and Easy

We highly suggest that you run all the tests before pushing a significant commit. It would be painful to copy/paste all of those lines above - so we've compiled a test script in the *utils* folder.

Just run this from the root of the repository:

```
./utils/run_tests.sh
```

It will execute all of the tests that would normally be run during the TravisCI build. If you want to see if it will pass TravisCI, you can just run that quick command on your machine.

### 11.3.4 Running the Test Suite

We use Nose for unit testing. All tests are placed in the `tests` folder.

- Just run the following:

```
nosetests -v
```

- Alternatively, you can use *invoke*, as mentioned above:

```
invoke unit.nose
```

Output:

```
test_overrides_yml_config: Tests the format of the overrides yml file for the RAM␣
↪service ... ok
test_passing_overall_iam_action_override: Tests iam:CreateAccessKey ... ok
test_get_dependent_actions_double (test_actions.ActionsTestCase) ... ok
test_get_dependent_actions_several (test_actions.ActionsTestCase) ... ok
test_get_dependent_actions_single (test_actions.ActionsTestCase) ... ok
test_analyze_by_access_level: Test out calling this as a library ... ok
test_determine_risky_actions_from_list: Test comparing requested actions to a list of␣
↪risky actions ... ok
test_get_actions_from_policy: Verify that the get_actions_from_policy function is␣
↪grabbing the actions ... ok
test_get_actions_from_policy_file_with_explicit_actions: Verify that we can get a␣
↪list of actions from a ... ok
test_get_actions_from_policy_file_with_wildcards: Verify that we can read the actions␣
↪from a file, ... ok
test_remove_actions_not_matching_access_level: Verify remove_actions_not_matching_
↪access_level is working as expected ... ok
test_get_findings: Ensure that finding.get_findings() combines two risk findings for␣
↪one policy properly. ... ok
test_get_findings_by_policy_name: Testing out the 'Findings' object ... ok
test_add_s3_permissions_management_arn (test_arn_action_group.ArnActionGroupTestCase)␣
↪... ok
test_get_policy_elements (test_arn_action_group.ArnActionGroupTestCase) ... ok
test_update_actions_for_raw_arn_format (test_arn_action_group.ArnActionGroupTestCase)␣
↪... ok
test_does_arn_match_case_1 (test_arns.ArnsTestCase) ... ok
test_does_arn_match_case_2 (test_arns.ArnsTestCase) ... ok
test_does_arn_match_case_4 (test_arns.ArnsTestCase) ... ok
test_does_arn_match_case_5 (test_arns.ArnsTestCase) ... ok
test_does_arn_match_case_6 (test_arns.ArnsTestCase) ... ok
test_does_arn_match_case_bucket (test_arns.ArnsTestCase) ... ok
test_determine_actions_to_expand: provide expanded list of actions, like ecr:* ... ok
test_minimize_statement_actions (test_minimize_wildcard_actions.
↪MinimizeWildcardActionsTestCase) ... ok
test_get_action_data: Tests function that gets details on a specific IAM Action. ...␣
↪ok
test_get_actions_at_access_level_that_support_wildcard_arns_only: Test function that␣
↪gets a list of ... ok
test_get_actions_for_service: Tests function that gets a list of actions per AWS␣
↪service. ... ok
test_get_actions_matching_condition_crud_and_arn: Get a list of IAM Actions matching␣
↪condition key, ... ok
test_get_actions_matching_condition_crud_and_wildcard_arn: Get a list of IAM Actions␣
↪matching condition key ... ok
test_get_actions_matching_condition_key: Tests a function that gathers all instances␣
↪in ... ok
test_get_actions_that_support_wildcard_arns_only: Tests function that shows all ... ok
test_get_actions_with_access_level: Tests function that gets a list of actions in a ..
↪. ok
test_get_actions_with_arn_type_and_access_level: Tests a function that gets a list of␣
↪... ok
test_get_all_actions_with_access_level: Get all actions with a given access level ...␣
↪ok
test_get_arn_type_details: Tests function that grabs details about a specific ARN␣
↪name ... ok
test_get_arn_types_for_service: Tests function that grabs arn_type and raw_arn pairs .
↪.. ok
```

(continues on next page)

(continued from previous page)

```
test_get_condition_key_details: Tests function that grabs details about a specific␣
↪condition key ... ok
test_get_condition_keys_for_service: Tests function that grabs a list of condition␣
↪keys per service. ... ok
test_get_raw_arns_for_service: Tests function that grabs a list of raw ARNs per␣
↪service ... ok
test_remove_actions_that_are_not_wildcard_arn_only: Tests function that removes␣
↪actions from a list that ... ok
test_actions_template (test_template.TemplateTestCase) ... ok
test_crud_template (test_template.TemplateTestCase) ... ok
test_actions_schema: Validates that the user-supplied YAML is working for CRUD mode ..
↪. ok
test_actions_schema: Validates that the user-supplied YAML is working for CRUD mode ..
↪. ok
test_print_policy_with_actions_having_dependencies (test_write_policy.
↪WritePolicyActionsTestCase) ... ok
test_write_policy (test_write_policy.WritePolicyCrudTestCase) ... ok
test_write_policy_beijing: Tests ARNs with the partiion `aws-cn` instead of just␣
↪`aws` ... ok
test_write_policy_govcloud: Tests ARNs with the partition `aws-us-gov` instead of␣
↪`aws` ... ok
test_wildcard_when_not_necessary: Attempts bypass of CRUD mode wildcard-only ... ok
test_write_actions_policy_with_library_only: Write an actions mode policy without␣
↪using the command line at all (library only) ... ok
test_write_crud_policy_with_library_only: Write an actions mode policy without using␣
↪the command line at all (library only) ... ok
test_actions_missing_actions: write-policy actions if the actions block is missing ...
↪ ok
test_allow_missing_access_level_categories_in_cfg: write-policy when the YAML file ...
↪ ok
test_allow_empty_access_level_categories_in_cfg: If the content of a list is an empty␣
↪string, it should sysexit ... ok
test_actions_missing_arn: write-policy actions command when YAML file block is␣
↪missing an ARN ... ok
test_actions_missing_description: write-policy when the YAML file is missing a␣
↪description ... ok
test_actions_missing_name: write-policy when the YAML file is missing a name ... ok

Ran 57 tests in 2.694s

OK
```

## 11.4 Project Structure

We'll focus mostly on the intent and approach of the major files (and subfolders) within the `policy_sentry/shared` directory:

### 11.4.1 Subfolders

**Folders per command**:

- The folders are mostly specific to their commands. For example, consider the files in the *policy_sentry/analysis* folder.

- **The files in this folder are specific to the *analyze* command**
  - They all can import from the *util* folder and the *shared* folder.
  - The files in this folder **don't import from other subfolders specific to other commands**, like *writing* or *downloading*. (*Note: There is an occasional exception here of re-using functions from the ‘querying‘ folder*)
  - Files in the *analysis* folder, to the *analyze* command. They don't import from each other, with the occasional exception of re-using functions from the *querying* folder. They all import common methods from the *util* folder and the *shared* folder as well.

**Files**:

- `shared/data/aws.sqlite3`: This is the pre-bundled IAM database. Third party packages can easily query the pre-bundled IAM database by connecting to the database like this: *db_session = connect_db(‘bundled’)*
- `shared/data/audit/*.txt`: These text files are the pre-bundled audit files that you can use with the `analyze-iam-policy` command. Currently they are limited to privilege escalation and resource exposure. For more information, see the page on Analyzing IAM Policies.
- `shared/data/docs/*.html`: These are HTML files wget'd from the Actions, Resources, and Condition Keys AWS documentation. This is used to build our database.
- *shared/data/access-level-overrides.yml*: This is created to override the access levels that AWS incorrectly states in their documentation. For instance, quite often, their service teams will say that an IAM action is “Tagging” when it really should be “Write” - for example, *secretsmanager:CreateSecret*.

### 11.4.2 Files and functions

**TODO: Generate documentation automagically based on docstrings**

## 11.5 Versioning

We try to follow Semantic Versioning as much as possible.

### 11.5.1 Version bumps

Just edit the *policy_sentry/bin/policy_sentry* file and update the __*version*__ variable:

```
#! /usr/bin/env python
"""
    policy_sentry is a tool for generating least-privilege IAM Policies.
"""
__version__ = '0.6.3'  # EDIT THIS
```

The *setup.py* file will automatically pick up the new version from that file for the package info. The *@click.version_option* decorator will also pick that up for the command line.

## 11.6 Roadmap

### 11.6.1 Condition Keys

Currently, Condition Keys are not supported by this script. For an example, see the KMS key Condition Key Table here. Note: The database does create a table of condition keys in case we develop future support for it, but it isn't used yet.

### 11.6.2 Log-based policy generation

We are considering building functionality to:

- Use Amazon Athena to query CloudTrail logs from an S3 bucket for AWS IAM API calls, similar to CloudTracker.
- **Instead of identifying the exact AWS IAM actions that were used, as CloudTracker currently does, we identify:**
    - Resource ARNs
    - Actions that indicate a CRUD level corresponding to that resource ARN. For example, if read access is granted to an S3 bucket folder path, assume all Read actions are needed for that folder path. Otherwise, we run into issues where CloudTrail actions and IAM actions don't match, which is a well documented issue by CloudTracker.
- **Query the logs to determine which principals touch which ARNs.**
    - For each IAM principal, create a list of ARNs.
    - For each ARN, plug that ARN into a policy_sentry yml file, and determine the CRUD level based on a lazy comparison of the action listed in the cloudtrail log vs the resource ARN.
    - And then run the policy_sentry yml file to generate an IAM policy that would have worked.

This was discussed in the original Hacker News post..

# CHAPTER 12

# Library Usage

Policy Sentry can be used as a Python library. Check out this documentation for more information and examples.

## 12.1 Getting Started with the Library

When using Policy Sentry manually, you have to build a local database file with the initialize function.

However, if you are developing your own Python code and you want to import Policy Sentry as a third party package, you can skip the initialization and leverage the local database file that is bundled with the Python package itself.

This is especially useful for developers who wish to leverage Policy Sentry's capabilities that require the use of the IAM database (such as querying the IAM database table). This way, you don't have to initialize the database and can just query it immediately.

The code example is located here. It is also shown below.

We've built a trick into the *connect_db* function that developers can specify to leverage the local database. The trick is to just use *'bundled'* as the single parameter for the *connect_db* method. See the example.

```
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_actions_for_service

def example():
    db_session = connect_db('bundled')  # This is the critical line. You just need to␣
↪specify `'bundled'` as the parameter.
    actions = get_actions_for_service(db_session, 'cloud9')  # Then you can leverage␣
↪any method that requires access to the database.
    for action in actions:
        print(action)

if __name__ == '__main__':
    example()
```

Try running the code from the root of the repository:

```
./examples/library-usage/example.py
```

The results will look like this:

```
cloud9:createenvironmentec2
cloud9:createenvironmentmembership
cloud9:deleteenvironment
cloud9:deleteenvironmentmembership
cloud9:describeenvironmentmemberships
cloud9:describeenvironmentstatus
cloud9:describeenvironments
cloud9:getusersettings
cloud9:listenvironments
cloud9:updateenvironment
cloud9:updateenvironmentmembership
cloud9:updateusersettings
```

## 12.2 Examples

These are examples for the modules and functions that will be of interest for developers leveraging Policy Sentry as a Python library.

### 12.2.1 Querying the IAM Database

The following are examples of how to leverage some of the functions available from Policy Sentry. The functions selected are likely to be of most interest to other developers.

These ones relate to querying the IAM database.

#### All

#### querying.all.get_all_services

```python
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.all import get_all_service_prefixes


if __name__ == '__main__':
    db_session = connect_db('bundled')
    all_service_prefixes = get_all_service_prefixes(db_session)
    print(all_service_prefixes)

"""
Output:

A list of every service prefix (like 'kms' or 's3') available in the IAM database.
Note that this will not include services that do not support any ARN types, like AWS␣
↪IQ.
"""
```

### querying.all.get_all_actions

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.all import get_all_actions


if __name__ == '__main__':
    db_session = connect_db('bundled')
    all_actions = get_all_actions(db_session)
    print(all_actions)

"""
Output:

Every IAM action available across all services, without duplicates
"""
```

## Actions

### querying.actions.get_action_data

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_action_data
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_action_data(db_session, 'ram', 'createresourceshare')
    print(json.dumps(output, indent=4))

"""
Output:

{
    'ram': [
        {
            'action': 'ram:createresourceshare',
            'description': 'Create resource share with provided resource(s) and/or␣
↪principal(s)',
            'access_level': 'Permissions management',
            'resource_arn_format': 'arn:${Partition}:ram:${Region}:${Account}
↪:resource-share/${ResourcePath}',
            'condition_keys': [
                'ram:RequestedResourceType',
                'ram:ResourceArn',
                'ram:RequestedAllowsExternalPrincipals'
            ],
            'dependent_actions': None
        },
        {
            'action': 'ram:createresourceshare',
            'description': 'Create resource share with provided resource(s) and/or␣
↪principal(s)',
```

(continues on next page)

(continued from previous page)

```
            'access_level': 'Permissions management',
            'resource_arn_format': '*',
            'condition_keys': [
                'aws:RequestTag/${TagKey}',
                'aws:TagKeys'
            ],
            'dependent_actions': None
        }
    ]
}
"""
```

### querying.actions.get_actions_for_service

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_actions_for_service
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_actions_for_service(db_session, 'cloud9')
    print(json.dumps(output, indent=4))

"""
Output:

[
    'ram:acceptresourceshareinvitation',
    'ram:associateresourceshare',
    'ram:createresourceshare',
    'ram:deleteresourceshare',
    'ram:disassociateresourceshare',
    'ram:enablesharingwithawsorganization',
    'ram:rejectresourceshareinvitation',
    'ram:updateresourceshare'
]
"""
```

### querying.actions.get_actions_matching_condition_key

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_actions_matching_condition_key
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_actions_matching_condition_key(db_session, "ses",
↪"ses:FeedbackAddress")
    print(json.dumps(output, indent=4))
```

(continues on next page)

(continued from previous page)

```
"""
Output:

[
    'ses:sendemail',
    'ses:sendbulktemplatedemail',
    'ses:sendcustomverificationemail',
    'ses:sendemail',
    'ses:sendrawemail',
    'ses:sendtemplatedemail'
]
"""
```

### querying.actions.get_actions_supporting_wilcards_only

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_actions_matching_condition_key
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_actions_matching_condition_key(db_session, "ses",
↪"ses:FeedbackAddress")
    print(json.dumps(output, indent=4))

"""
Output:

[
    'ses:sendemail',
    'ses:sendbulktemplatedemail',
    'ses:sendcustomverificationemail',
    'ses:sendemail',
    'ses:sendrawemail',
    'ses:sendtemplatedemail'
]
"""
```

### querying.actions.get_actions_with_access_levels

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_actions_with_access_level
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_actions_with_access_level(db_session, 's3', 'Permissions management')
    print(json.dumps(output, indent=4))

"""
```

(continues on next page)

(continued from previous page)

```
Output:

    s3:bypassgovernanceretention
    s3:deleteaccesspointpolicy
    s3:deletebucketpolicy
    s3:objectowneroverridetobucketowner
    s3:putaccesspointpolicy
    s3:putaccountpublicaccessblock
    s3:putbucketacl
    s3:putbucketpolicy
    s3:putbucketpublicaccessblock
    s3:putobjectacl
    s3:putobjectversionacl
"""
```

### querying.actions.get_actions_with_arn_type_and_access_level

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_actions_with_arn_type_and_access_level
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_actions_with_arn_type_and_access_level(db_session, "ram", "resource-
↪share", "Permissions management")
    print(json.dumps(output, indent=4))

"""
Output:

[
    'ram:associateresourceshare',
    'ram:createresourceshare',
    'ram:deleteresourceshare',
    'ram:disassociateresourceshare',
    'ram:updateresourceshare'
]
"""
```

### querying.actions.get_dependent_actions

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.actions import get_dependent_actions
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_dependent_actions(db_session, ["ec2:associateiaminstanceprofile"])
    print(json.dumps(output, indent=4))

```

(continues on next page)

(continued from previous page)

```
"""
Output:

[
    "iam:passrole"
]
"""
```

### ARNs

#### querying.arns.get_arn_type_details

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.arns import get_arn_type_details
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_arn_type_details(db_session, "cloud9", "environment")
    print(json.dumps(output, indent=4))

"""
Output:

{
    "resource_type_name": "environment",
    "raw_arn": "arn:${Partition}:cloud9:${Region}:${Account}:environment:${ResourceId}
↪",
    "condition_keys": None
}
"""
```

#### querying.arns.get_arn_types_for_service

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.arns import get_arn_types_for_service
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_arn_types_for_service(db_session, "s3")
    print(json.dumps(output, indent=4))

"""
Output:

{
    "accesspoint": "arn:${Partition}:s3:${Region}:${Account}:accesspoint/$
↪{AccessPointName}",
    "bucket": "arn:${Partition}:s3:::${BucketName}",
```

(continues on next page)

(continued from previous page)

```
    "object": "arn:${Partition}:s3:::${BucketName}/${ObjectName}",
    "job": "arn:${Partition}:s3:${Region}:${Account}:job/${JobId}",
}
"""
```

### querying.arns.get_raw_arns_for_service

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.arns import get_raw_arns_for_service
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_raw_arns_for_service(db_session, "s3")
    print(json.dumps(output, indent=4))

"""
Output:

[
    "arn:${Partition}:s3:${Region}:${Account}:accesspoint/${AccessPointName}",
    "arn:${Partition}:s3:::${BucketName}",
    "arn:${Partition}:s3:::${BucketName}/${ObjectName}",
    "arn:${Partition}:s3:${Region}:${Account}:job/${JobId}"
]
"""
```

## Conditions

### querying.conditions.get_condition_key_details

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.conditions import get_condition_key_details
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_condition_key_details(db_session, "cloud9", "cloud9:Permissions")
    print(json.dumps(output, indent=4))

"""
Output:

{
    "name": "cloud9:Permissions",
    "description": "Filters access by the type of AWS Cloud9 permissions",
    "condition_value_type": "string"
}
"""
```

### querying.conditions.get_condition_keys_for_service

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.querying.conditions import get_condition_keys_for_service
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    output = get_condition_keys_for_service(db_session, "cloud9")
    print(json.dumps(output, indent=4))

"""
Output:

[
    'cloud9:EnvironmentId',
    'cloud9:EnvironmentName',
    'cloud9:InstanceType',
    'cloud9:Permissions',
    'cloud9:SubnetId',
    'cloud9:UserArn'
]
"""
```

## 12.2.2 Writing Policies

The following are examples of how to leverage some of the functions available from Policy Sentry. The functions selected are likely to be of most interest to other developers.

These ones refer to leveraging Policy Sentry as a library to write IAM policies.

### Actions Mode: Writing Policies by providing a list of Actions

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.writing.template import get_actions_template_dict
from policy_sentry.command.write_policy import write_policy_with_template
import json


if __name__ == '__main__':
    db_session = connect_db('bundled')
    actions_template = get_actions_template_dict()
    actions_to_add = ['kms:CreateGrant', 'kms:CreateCustomKeyStore',
↪'ec2:AuthorizeSecurityGroupEgress',
                      'ec2:AuthorizeSecurityGroupIngress']
    actions_template['actions'].extend(actions_to_add)
    policy = write_policy_with_template(db_session, actions_template)
    print(json.dumps(policy, indent=4))

"""
Output:
```

(continues on next page)

(continued from previous page)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "KmsPermissionsmanagementKmskey",
            "Effect": "Allow",
            "Action": [
                "kms:creategrant"
            ],
            "Resource": [
                "arn:${Partition}:kms:${Region}:${Account}:key/${KeyId}"
            ]
        },
        {
            "Sid": "Ec2WriteSecuritygroup",
            "Effect": "Allow",
            "Action": [
                "ec2:authorizesecuritygroupegress",
                "ec2:authorizesecuritygroupingress"
            ],
            "Resource": [
                "arn:${Partition}:ec2:${Region}:${Account}:security-group/$
↪{SecurityGroupId}"
            ]
        },
        {
            "Sid": "MultMultNone",
            "Effect": "Allow",
            "Action": [
                "kms:createcustomkeystore",
                "cloudhsm:describeclusters"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
"""
```

### CRUD Mode: Writing Policies by Access Levels and ARNs

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.writing.template import get_crud_template_dict
from policy_sentry.command.write_policy import write_policy_with_template
import json


if __name__ == '__main__':
    db_session = connect_db('bundled')
    crud_template = get_crud_template_dict()
    wildcard_actions_to_add = ["kms:createcustomkeystore", "cloudhsm:describeclusters
↪"]
    crud_template['mode'] = 'crud'
```

(continues on next page)

(continued from previous page)

```
    crud_template['read'].append("arn:aws:secretsmanager:us-east-
↪1:123456789012:secret:mysecret")
    crud_template['write'].append("arn:aws:secretsmanager:us-east-
↪1:123456789012:secret:mysecret")
    crud_template['list'].append("arn:aws:s3:::example-org-sbx-vmimport/stuff")
    crud_template['permissions-management'].append("arn:aws:kms:us-east-
↪1:123456789012:key/123456")
    crud_template['wildcard'].extend(wildcard_actions_to_add)
    crud_template['tagging'].append("arn:aws:ssm:us-east-1:123456789012:parameter/test
↪")
    # Modify it
    policy = write_policy_with_template(db_session, crud_template)
    print(json.dumps(policy, indent=4))


"""
Output:

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "MultMultNone",
            "Effect": "Allow",
            "Action": [
                "kms:createcustomkeystore"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "SecretsmanagerReadSecret",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:describesecret",
                "secretsmanager:getresourcepolicy",
                "secretsmanager:getsecretvalue",
                "secretsmanager:listsecretversionids"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
            ]
        },
        {
            "Sid": "SecretsmanagerWriteSecret",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:cancelrotatesecret",
                "secretsmanager:deletesecret",
                "secretsmanager:putsecretvalue",
                "secretsmanager:restoresecret",
                "secretsmanager:rotatesecret",
                "secretsmanager:updatesecret",
                "secretsmanager:updatesecretversionstage"
            ],
            "Resource": [
```

(continues on next page)

(continued from previous page)

```
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
            ]
        },
        {
            "Sid": "KmsPermissionsmanagementKmskey",
            "Effect": "Allow",
            "Action": [
                "kms:creategrant",
                "kms:putkeypolicy",
                "kms:retiregrant",
                "kms:revokegrant"
            ],
            "Resource": [
                "arn:aws:kms:us-east-1:123456789012:key/123456"
            ]
        }
    ]
}
"""
```

### 12.2.3 Analyzing Policies

The following are examples of how to leverage some of the functions available from Policy Sentry. The functions selected are likely to be of most interest to other developers.

These ones relate to the analysis features.

#### Analyzing by access level

Determine if a policy has any actions with a given access level. This is particularly useful when determining who has ‘Permissions management’ level access.

#### analysis.analyze_by_access_level

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.analysis.analyze import analyze_by_access_level
import json

if __name__ == '__main__':
    db_session = connect_db('bundled')
    permissions_management_policy = {
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": [
                    # These ones are Permissions management
                    "ecr:SetRepositoryPolicy",
                    "secretsmanager:DeleteResourcePolicy",
                    "iam:UpdateAccessKey",
                    # These ones are not permissions management
```

(continues on next page)

(continued from previous page)

```
                    "ecr:GetRepositoryPolicy",
                    "ecr:DescribeRepositories",
                    "ecr:ListImages",
                    "ecr:DescribeImages",
                ],
                "Resource": "*"
            }
        ]
    }
    permissions_management_actions = analyze_by_access_level(db_session, permissions_
↪management_policy, "permissions-management")
    print(json.dumps(permissions_management_actions, indent=4))

"""
Output:

[
    'ecr:setrepositorypolicy',
    'iam:updateaccesskey',
    'secretsmanager:deleteresourcepolicy'
]
"""
```

### Expanding actions from a policy file

```
#!/usr/bin/env python
from policy_sentry.shared.database import connect_db
from policy_sentry.util.policy_files import get_actions_from_policy
from policy_sentry.analysis.analyze import determine_actions_to_expand
import json

POLICY_JSON_TO_EXPAND = {
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "cloud9:*",
      ],
      "Resource": "*"
    }
  ]
}


if __name__ == '__main__':
    db_session = connect_db('bundled')
    requested_actions = get_actions_from_policy(POLICY_JSON_TO_EXPAND)
    expanded_actions = determine_actions_to_expand(db_session, requested_actions)
    print(json.dumps(expanded_actions, indent=4))


"""
Output:
```

(continues on next page)

(continued from previous page)

```
[
    "cloud9:createenvironmentec2",
    "cloud9:createenvironmentmembership",
    "cloud9:deleteenvironment",
    "cloud9:deleteenvironmentmembership",
    "cloud9:describeenvironmentmemberships",
    "cloud9:describeenvironments",
    "cloud9:describeenvironmentstatus",
    "cloud9:getusersettings",
    "cloud9:listenvironments",
    "cloud9:updateenvironment",
    "cloud9:updateenvironmentmembership",
    "cloud9:updateusersettings"
]
"""
```

## 12.3 Module Reference

These are modules and functions that will be of interest for developers leveraging Policy Sentry as a Python library.

### 12.3.1 Querying

#### querying.all

IAM Database queries that are not specific to either the Actions, ARNs, or Condition Keys tables.

`policy_sentry.querying.all.`**`get_all_actions`**(*db_session*)

Gets a huge list of all IAM actions. This is used as part of the policyuniverse approach to minimizing IAM Policies to meet AWS-mandated character limits on policies.

**Parameters** **`db_session`** – SQLAlchemy database session object

**Returns** A list of all actions present in the database.

`policy_sentry.querying.all.`**`get_all_service_prefixes`**(*db_session*)

Gets all the AWS service prefixes from the actions table.

If the action table does NOT have specific IAM actions (and therefore only supports * actions), then it will not be included in the response.

**Parameters** **`db_session`** – The SQLAlchemy database session

**Returns** A list of all AWS service prefixes present in the table.

#### querying.actions

Methods that execute specific queries against the SQLite database for the ACTIONS table. This supports the policy_sentry query functionality

`policy_sentry.querying.actions.`**`get_action_data`**(*db_session*, *service*, *name*)

Get details about an IAM Action in JSON format.

**Parameters**

- **`db_session`** – SQLAlchemy database session object

- **`service`** – An AWS service prefix, like *s3* or *kms*
- **`name`** – The name of an AWS IAM action, like *GetObject*. To get data about all actions in a service, specify "*"

**Returns** A dictionary containing metadata about an IAM Action.

`policy_sentry.querying.actions.`**`get_actions_at_access_level_that_support_wildcard_arns_only`**

Get a list of actions at an access level that do not support restricting the action to resource ARNs.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – A single AWS service prefix, like *s3* or *kms*
- **`access_level`** – An access level as it is written in the database, such as 'Read', 'Write', 'List', 'Permisssions management', or 'Tagging'

**Returns** A list of actions

`policy_sentry.querying.actions.`**`get_actions_for_service`**(*db_session*, *service*)

Get a list of available actions per AWS service

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – An AWS service prefix, like *s3* or *kms*

**Returns** A list of actions

`policy_sentry.querying.actions.`**`get_actions_matching_condition_crud_and_arn`**(*db_session*, *condition_key*, *access_level*, *raw_arn*)

Get a list of IAM Actions matching a condition key, CRUD level, and raw ARN format.

**Parameters**

- **`db_session`** – SQL Alchemy database session
- **`condition_key`** – A condition key, like aws:TagKeys
- **`access_level`** – Access level that matches the database value. "Read", "Write", "List", "Tagging", or "Permissions management"
- **`raw_arn`** – The raw ARN format in the database, like arn:${Partition}:s3:::${BucketName}

**Returns** List of IAM Actions

`policy_sentry.querying.actions.`**`get_actions_matching_condition_key`**(*db_session*, *service*, *condition_key*)

Get a list of actions under a service that allow the use of a specified condition key

**Parameters**

- **`db_session`** – SQLAlchemy database session
- **`service`** – A single AWS service prefix
- **`condition_key`** – The condition key to look for.

**Returns** A list of actions

`policy_sentry.querying.actions.`**`get_actions_that_support_wildcard_arns_only`**(*db_session*, *service*)

Get a list of actions that do not support restricting the action to resource ARNs.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – A single AWS service prefix, like *s3* or *kms*

**Returns** A list of actions

`policy_sentry.querying.actions.`**`get_actions_with_access_level`**(*db_session*, *service*, *access_level*)

Get a list of actions in a service under different access levels.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – A single AWS service prefix, like *s3* or *kms*
- **`access_level`** – An access level as it is written in the database, such as 'Read', 'Write', 'List', 'Permisssions management', or 'Tagging'

**Returns** A list of actions

`policy_sentry.querying.actions.`**`get_actions_with_arn_type_and_access_level`**(*db_session*, *service*, *resource_type_name*, *access_level*)

Get a list of actions in a service under different access levels, specific to an ARN format.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – A single AWS service prefix, like *s3* or *kms*
- **`resource_type_name`** – The ARN type name, like *bucket* or *key*

**Returns** A list of actions

`policy_sentry.querying.actions.`**`get_dependent_actions`**(*db_session*, *actions_list*)

Given a list of IAM Actions, query the database to determine if the action has dependent actions in the fifth column of the Resources, Actions, and Condition keys tables. If it does, add the dependent actions to the list, and return the updated list.

It includes the original action in there as well. So, if you supply kms:CreateCustomKeyStore, it will give you kms:CreateCustomKeyStore as well as cloudhsm:DescribeClusters

To get dependent actions for a single given IAM action, just provide the action as a list with one item, like this: get_dependent_actions(db_session, ['kms:CreateCustomKeystore'])

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`actions_list`** – A list of actions to use in querying the database for dependent actions

**Returns** Updated list of actions, including dependent actions if applicable.

`policy_sentry.querying.actions.`**`remove_actions_not_matching_access_level`**(*db_session*, *actions_list*, *access_level*)

Given a list of actions, return a list of actions that match an access level

**Parameters**

- **`db_session`** – The SQLAlchemy database session
- **`actions_list`** – A list of actions
- **`access_level`** – 'read', 'write', 'list', 'tagging', or 'permissions-management'

**Returns** Updated list of actions, where the actions not matching the requested access level are removed.

`policy_sentry.querying.actions.`**`remove_actions_that_are_not_wildcard_arn_only`**(*db_session*, *actions_list*)

Given a list of actions, remove the ones that CAN be restricted to ARNs, leaving only the ones that cannot.

**Parameters**

- **`db_session`** – SQL Alchemy database session object
- **`actions_list`** – A list of actions

**Returns** An updated list of actions

**Return type** list

### querying.arns

Methods that execute specific queries against the SQLite database for the ARN table. This supports the policy_sentry query functionality

`policy_sentry.querying.arns.`**`get_arn_data`**(*db_session*, *service*, *name*)

Get details about ARNs in JSON format.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – An AWS service prefix, like *s3* or *kms*
- **`name`** – The name of a resource type, like *bucket* or *object*. To get details on ALL arns in a service, specify "*" here.

**Returns** Metadata about an ARN type

`policy_sentry.querying.arns.`**`get_arn_type_details`**(*db_session*, *service*, *name*)

Get details about a resource ARN type name in JSON format.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – An AWS service prefix, like *s3* or *kms*

- **`name`** – The name of a resource type, like *bucket* or *object*

**Returns** Metadata about an ARN type

`policy_sentry.querying.arns.`**`get_arn_types_for_service`**(*db_session*, *service*)

Get a list of available ARN short names per AWS service.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – An AWS service prefix, like *s3* or *kms*

**Returns** A list of ARN types, like *bucket* or *object*

`policy_sentry.querying.arns.`**`get_raw_arns_for_service`**(*db_session*, *service*)

Get a list of available raw ARNs per AWS service

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – An AWS service prefix, like *s3* or *kms*

**Returns** A list of raw ARNs

`policy_sentry.querying.arns.`**`get_resource_type_name_with_raw_arn`**(*db_session*, *raw_arn*)

Given a raw ARN, return the resource type name as shown in the database.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`raw_arn`** – The raw ARN stored in the database, like ‘arn:${Partition}:s3:::${BucketName}’

**Returns** The resource type name, like bucket

### querying.conditions

Methods that execute specific queries against the SQLite database for the CONDITIONS table. This supports the policy_sentry query functionality

`policy_sentry.querying.conditions.`**`get_condition_key_details`**(*db_session*, *service*, *condition_key_name*)

Get details about a specific condition key in JSON format

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – An AWS service prefix, like *ec2* or *kms*
- **`condition_key_name`** – The name of a condition key, like *ec2:Vpc*

**Returns** Metadata about the condition key

`policy_sentry.querying.conditions.`**`get_condition_keys_available_to_raw_arn`**(*db_session*, *raw_arn*)

Get a list of condition keys available to a RAW ARN

**Parameters**

- **`db_session`** – SQLAlchemy database session object

- **`raw_arn`** – The value in the database, like arn:${Partition}:s3:::${BucketName}/${ObjectName}

`policy_sentry.querying.conditions.`**`get_condition_keys_for_service`**(*db_session*, *service*)

Get a list of available conditions per AWS service

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`service`** – An AWS service prefix, like *s3* or *kms*

**Returns** A list of condition keys

`policy_sentry.querying.conditions.`**`get_condition_value_type`**(*db_session*, *condition_key*)

Get the data type of the condition key - like Date, String, etc. :param db_session: SQLAlchemy database session object :param condition_key: A condition key, like a4b:filters_deviceType :return:

`policy_sentry.querying.conditions.`**`get_conditions_for_action_and_raw_arn`**(*db_session*, *action*, *raw_arn*)

Get a list of conditions available to an action.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`action`** – The IAM action, like s3:GetObject
- **`raw_arn`** – The raw ARN format specific to the action

**Returns**

### 12.3.2 Writing

#### command.write_policy

Given a Policy Sentry YML template, write a least-privilege IAM Policy in CRUD mode or Actions mode.

`policy_sentry.command.write_policy.`**`write_policy_with_template`**(*db_session*, *cfg*, *minimize=None*)

This function is called by write-policy so the config can be passed in as a dict without running into a Click-related error. Use this function, rather than the write-policy function, if you are using Policy Sentry as a python library.

**Parameters**

- **`db_session`** – SQL Alchemy database session object
- **`cfg`** – The loaded YAML as a dict. Must follow Policy Sentry dictated format.
- **`minimize`** – Minimize the resulting statement with *safe* usage of wildcards to reduce policy length. Set this to the character length you want - for example, 0, or 4. Defaults to none.

#### writing.sid_group

sid_group indicates that this is a collection of policy-related data organized by their SIDs

**class** `policy_sentry.writing.sid_group.`**`SidGroup`**

This class is critical to the creation of least privilege policies. It uses the SIDs as namespaces. The namespaces follow this format:

> {Servicename}{Accesslevel}{Resourcetypename}

So, a resulting statement's SID might look like 'S3ListBucket'

If a condition key is supplied (like s3:RequestJob), the SID string will be significantly longer. It will resemble this format:

> {Servicename}{Accesslevel}{Resourcetypename}{Conditionkeystring}{Conditiontypestring}{Conditionkeyvalue}

**For example: EC2 write actions on the security-group resource, using the following condition map:**

> **"Condition": {** "StringEquals": {"ec2:ResourceTag/Owner": "${aws:username}"}
>
> }

**The resulting SID would be:** Ec2WriteSecuritygroupResourcetagownerStringequalsAwsusername

**Or, for actions that support wildcard ARNs only, an example could be:** Ec2WriteMultResourcetagownerStringequalsAwsuse

**`add_action_without_resource_constraint`**(*action*)

This handles the cases where certain actions do not handle resource constraints - either by AWS, or for flexibility when adding dependent actions.

**Parameters** **`action`** – The single action to add to the MultMultNone SID namespace. For instance, s3:ListAllMyBuckets

**`add_by_arn_and_access_level`**(*db_session*, *arn_list*, *access_level*, *conditions_block=None*)

This adds the user-supplied ARN(s), service prefixes, access levels, and condition keys (if applicable) given by the user. It derives the list of IAM actions based on the user's requested ARNs and access levels.

**Parameters**

- **`db_session`** – SQLAlchemy database session
- **`arn_list`** – Just a list of resource ARNs.
- **`access_level`** – "Read", "List", "Tagging", "Write", or "Permissions management"
- **`conditions_block`** – Optionally, a condition block with one or more conditions

**`add_by_list_of_actions`**(*db_session*, *supplied_actions*)

Takes a list of actions, queries the database for corresponding arns, adds them to the object.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`supplied_actions`** – A list of supplied actions

**`get_rendered_policy`**(*db_session*, *minimize=None*)

Get the JSON rendered policy

**Parameters**

- **`db_session`** – SQLAlchemy database session
- **`minimize`** – Reduce the character count of policies without creating overlap with other action names

**Return type** dict

**get_sid**(*sid*)
Get a single group by the SID identifier

**get_sid_group**()
Get the whole SID group as JSON

**get_universal_conditions**()
Get the universal conditions maps back as a dict

**Return type** dict

**list_sids**()
Get a list of all of them by their identifiers

**Return type** list

**process_template**(*db_session*, *cfg*, *minimize=None*)
Process the Policy Sentry template as a dict. This auto-detects whether or not the file is in CRUD mode or Actions mode.

**Parameters**

- **db_session** – SQLAlchemy database session object
- **cfg** – The loaded YAML as a dict. Must follow Policy Sentry dictated format.
- **minimize** – Minimize the resulting statement with *safe* usage of wildcards to reduce policy length. Set this to the character length you want - for example, 0, or 4. Defaults to none.

**remove_actions_duplicated_in_wildcard_arn**()
Removes actions from the object that are in a resource-specific ARN, as well as the * resource. For example, if ssm:GetParameter is restricted to a specific parameter path, as well as *, then we want to remove the * option to force least privilege.

**remove_actions_not_matching_these**(*actions_to_keep*)

**Parameters** **actions_to_keep** – A list of actions to leave in the policy. All actions not in this list are removed.

policy_sentry.writing.sid_group.**create_policy_sid_namespace**(*service*, *access_level*, *resource_type_name*, *condition_block=None*)
Simply generates the SID name. The SID groups ARN types that share an access level.

For example, S3 objects vs. SSM Parameter have different ARN types - as do S3 objects vs S3 buckets. That's how we choose to group them.

**Parameters**

- **service** – "ssm"
- **access_level** – "Read"
- **resource_type_name** – "parameter"
- **condition_block** – {"condition_key_string": "ec2:ResourceTag/purpose", "condition_type_string": "StringEquals", "condition_value": "test"}

**Returns** SsmReadParameter

**Return type** str

`policy_sentry.writing.sid_group.`**`remove_actions_that_are_not_wildcard_arn_only`**`(`*db_session*, *actions_list*`)`

Given a list of actions, remove the ones that CAN be restricted to ARNs, leaving only the ones that cannot.

**Parameters**

- **`db_session`** – SQL Alchemy database session object
- **`actions_list`** – A list of actions

**Returns** An updated list of actions

**Return type** list

### writing.template

Templates for the policy_sentry YML files. These can be used for generating policies

`policy_sentry.writing.template.`**`create_actions_template`**`(`*name*`)`
Generate the Actions YML template with Jinja2

`policy_sentry.writing.template.`**`create_crud_template`**`(`*name*`)`
Generate the CRUD YML Template with Jinja2

`policy_sentry.writing.template.`**`get_actions_template_dict`**`()`
Get the Actions template in dict format.

`policy_sentry.writing.template.`**`get_crud_template_dict`**`()`
Generate the CRUD template in dict format

### writing.validate

Validation for the Policy Sentry YML Templates.

`policy_sentry.writing.validate.`**`check`**`(`*conf_schema*, *conf*`)`
Validates a user-supplied JSON vs a defined schema. :param conf_schema: The Schema object that defines the required structure. :param conf: The user-supplied schema to validate against the required structure.

`policy_sentry.writing.validate.`**`check_actions_schema`**`(`*cfg*`)`
Determines whether the user-provided config matches the required schema for Actions mode

`policy_sentry.writing.validate.`**`check_crud_schema`**`(`*cfg*`)`
Determines whether the user-provided config matches the required schema for CRUD mode

`policy_sentry.writing.validate.`**`validate_condition_block`**`(`*condition_block*`)`

**Parameters** **`condition_block`** – {“condition_key_string”: “ec2:ResourceTag/purpose”, “condition_type_string”: “StringEquals”, “condition_value”: “test”}

**Returns**

### writing.minimize

Functions for Minimizing statements, heavily borrowed from policyuniverse. https://github.com/Netflix-Skunkworks/policyuniverse/

IAM Policies have character limits, which apply to individual policies, and there are also limits on the total aggregate policy sizes. As such, it is not possible to use exhaustive list of explicit IAM actions. To have granular control of specific IAM policies, we must use wildcards on IAM Actions, only in a programmatic manner.

This is typically performed by humans by reducing policies to s3:Get*, ec2:Describe*, and other approaches of the sort.

Netflix's PolicyUniverse has address

https://aws.amazon.com/iam/faqs/ Q: How many policies can I attach to an IAM role? * For inline policies: You can add as many inline policies as you want to a user, role, or group, but

> the total aggregate policy size (the sum size of all inline policies) per entity cannot exceed the following limits: - User policy size cannot exceed 2,048 characters. - Role policy size cannot exceed 10,240 characters. - Group policy size cannot exceed 5,120 characters.

- For managed policies: You can add up to 10 managed policies to a user, role, or group.
- The size of each managed policy cannot exceed 6,144 characters.

`policy_sentry.writing.minimize.`**`check_min_permission_length`**(*permission*, *minchars=None*)

> Adapted version of policyuniverse's _check_permission_length. We are commenting out the skipping prefix message https://github.com/Netflix-Skunkworks/policyuniverse/blob/master/policyuniverse/expander_minimizer.py#L111

`policy_sentry.writing.minimize.`**`get_denied_prefixes_from_desired`**(*desired_actions*, *all_actions*)

> Adapted version of policyuniverse's _get_denied_prefixes_from_desired, here: https://github.com/Netflix-Skunkworks/policyuniverse/blob/master/policyuniverse/expander_minimizer.py#L101

`policy_sentry.writing.minimize.`**`minimize_statement_actions`**(*desired_actions*, *all_actions*, *minchars=None*)

> This is a condensed version of policyuniverse's minimize_statement_actions, changed for our purposes. https://github.com/Netflix-Skunkworks/policyuniverse/blob/master/policyuniverse/expander_minimizer.py#L123

### 12.3.3 Analyzing

#### analysis.analyze

Functions to support the analyze capability in this tool

`policy_sentry.analysis.analyze.`**`analyze_by_access_level`**(*db_session*, *policy_json*, *access_level*)

> Determine if a policy has any actions with a given access level. This is particularly useful when determining who has 'Permissions management' level access
>
> **Parameters**
>
> - **`db_session`** – SQLAlchemy database session
> - **`policy_json`** – a dictionary representing the AWS JSON policy
> - **`access_level`** – The normalized access level - either 'read', 'list', 'write', 'tagging', or 'permissions-management'

`policy_sentry.analysis.analyze.`**`analyze_policy_directory`**(*db_session*, *policy_directory*, *account_id*, *from_audit_file*, *finding_type*, *excluded_role_patterns*)

> Audits a directory of policy JSON files.
>
> **Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`policy_directory`** – The file directory where the policies are stored
- **`account_id`** – The AWS Account ID
- **`from_audit_file`** – The file containing the list of problematic actions
- **`finding_type`** – The type of finding - resource_exposure, privilege_escalation, network_exposure, or credentials_exposure

**Returns** A dictionary of findings with the policy names as keys.

`policy_sentry.analysis.analyze.`**`analyze_policy_file`**(*db_session*, *policy_file*, *account_id*, *from_audit_file*, *finding_type*, *excluded_role_patterns*)

Given a policy file, determine risky actions based on a separate file containing a list of actions. If it matches a policy exclusion pattern from the report-config.yml file, that policy file will be skipped.

**Parameters**

- **`db_session`** – SQLAlchemy database session object
- **`policy_file`** – The path to the policy file to be evaluated
- **`account_id`** – The AWS Account ID
- **`from_audit_file`** – The file containing the list of problematic actions
- **`finding_type`** – The type of finding - resource_exposure, privilege_escalation, network_exposure, or credentials_exposure
- **`excluded_role_patterns`** – A RegEx pattern for excluding policy names from evaluation.

**Returns** False if the policy name matches excluded role patterns, or if it does not, a dictionary containing the findings.

**Return type** dict

`policy_sentry.analysis.analyze.`**`analyze_statement_by_access_level`**(*db_session*, *statement_json*, *access_level*)

Determine if a statement has any actions with a given access level.

**Parameters**

- **`db_session`** – SQLAlchemy database session
- **`statement_json`** – a dictionary representing a statement from an AWS JSON policy
- **`access_level`** – The normalized access level - either ‘read’, ‘list’, ‘write’, ‘tagging’, or ‘permissions-management’

`policy_sentry.analysis.analyze.`**`determine_actions_to_expand`**(*db_session*, *action_list*)

Determine if an action needs to get expanded from its wildcard

**Parameters**

- **`db_session`** – A SQLAlchemy database session object
- **`action_list`** – A list of actions

**Returns** A list of actions

**Return type** list

`policy_sentry.analysis.analyze.`**`determine_risky_actions`**(*requested_actions*, *audit_file*)
compare the actions in the policy against the audit file of high risk actions

**Parameters**

- **`requested_actions`** – A list of the actions that are requested by the policy under evaluation
- **`audit_file`** – The absolute path to the file that contains a list of IAM action to evaluate.

**Returns** a list of any actions that are included in the file of risky actions

`policy_sentry.analysis.analyze.`**`determine_risky_actions_from_list`**(*requested_actions*, *risky_actions*)
compare the actions in the policy against a list of high risk actions

**Parameters**

- **`requested_actions`** – A list of the actions that are requested by the policy under evaluation
- **`risky_actions`** – A list of risky IAM actions to evaluate.

**Returns** a list of any actions that are included in the file of risky actions

`policy_sentry.analysis.analyze.`**`expand`**(*action*, *db_session*)
expand the action wildcards into a full action

**Parameters**

- **`action`** – An action in the form with a wildcard - like s3:Get*, or s3:L*
- **`db_session`** – SQLAlchemy database session object

**Returns** A list of all the expanded actions (like actions matching s3:Get*)

**Return type** list

`policy_sentry.analysis.analyze.`**`read_risky_iam_permissions_text_file`**(*audit_file*)
read in the audit file of high risk actions

**Parameters** **`audit_file`** – Path to the file containing a list of risky actions

**Return risky_actions** A list of actions from the file

### 12.3.4 Utilities

#### util.policy_files

A few methods for parsing policies.

`policy_sentry.util.policy_files.`**`get_actions_from_json_policy_file`**(*file*)
read the json policy file and return a list of actions

`policy_sentry.util.policy_files.`**`get_actions_from_policy`**(*data*)
Given a policy dictionary, create a list of the actions

`policy_sentry.util.policy_files.`**`get_actions_from_statement`**(*statement*)
Given a statement dictionary, create a list of the actions

### util.arns

Functions to use for parsing ARNs, matching ARN types, and getting the right fragment/component from an ARN string,

`policy_sentry.util.arns.`**`arn_has_colons`**`(`*arn*`)`

Given an ARN, determine if the ARN has colons in it. Just useful for the hacky methods for parsing ARN namespaces. See http://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html for more details on ARN namespacing.

`policy_sentry.util.arns.`**`arn_has_slash`**`(`*arn*`)`

Given an ARN, determine if the ARN has a stash in it. Just useful for the hacky methods for parsing ARN namespaces. See http://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html for more details on ARN namespacing.

`policy_sentry.util.arns.`**`does_arn_match`**`(`*arn_to_test*, *arn_in_database*`)`

Given two ARNs, determine if they match. The cases supported are outlined below.

Case 1: arn:partition:service:region:account-id:resource

Case 2: arn:partition:service:region:account-id:resourcetype/resource

Case 3: arn:partition:service:region:account-id:resourcetype/resource/qualifier

Case 4: arn:partition:service:region:account-id:resourcetype/resource:qualifier

Case 5: arn:partition:service:region:account-id:resourcetype:resource

Case 6: arn:partition:service:region:account-id:resourcetype:resource:qualifier

Source: https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html#genref-arns

**Parameters** **`arn`** – ARN to parse

**Returns** result of whether or not the ARNs match

`policy_sentry.util.arns.`**`get_account_from_arn`**`(`*arn*`)`

Given an ARN, return the account ID in the ARN, if it is available. In certain cases like S3 it is not

`policy_sentry.util.arns.`**`get_partition_from_arn`**`(`*arn*`)`

Given an ARN string, return the partition string. This is usually *aws* unless you are in C2S or AWS GovCloud.

`policy_sentry.util.arns.`**`get_region_from_arn`**`(`*arn*`)`

Given an ARN, return the region in the ARN, if it is available. In certain cases like S3 it is not

`policy_sentry.util.arns.`**`get_resource_from_arn`**`(`*arn*`)`

Given an ARN, parse it according to ARN namespacing and return the resource. See http://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html for more details on ARN namespacing.

`policy_sentry.util.arns.`**`get_resource_path_from_arn`**`(`*arn*`)`

Given an ARN, parse it according to ARN namespacing and return the resource path. See http://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html for more details on ARN namespacing.

`policy_sentry.util.arns.`**`get_service_from_arn`**`(`*arn*`)`

Given an ARN string, return the service

`policy_sentry.util.arns.`**`parse_arn`**`(`*arn*`)`

Given an ARN, split up the ARN into the ARN namespacing schema dictated by the AWS docs.

### util.file

Functions that relate to manipulating files, loading files, and managing filepaths.

`policy_sentry.util.file.`**`check_valid_file_path`**(*file*)

Checks if the file path is valid.

**Parameters** **`file`** – The file to check.

**Returns** True if it exists, False if it does not

**Return type** bool

`policy_sentry.util.file.`**`create_directory_if_it_doesnt_exist`**(*directory*)

Equivalent of mkdir -p

`policy_sentry.util.file.`**`list_files_in_directory`**(*directory*)

Equivalent of ls command, and return the list of files

`policy_sentry.util.file.`**`read_this_file`**(*filename*)

Read a file at a path and return the lines from each file

`policy_sentry.util.file.`**`read_yaml_file`**(*filename*)

Reads a YAML file, safe loads, and returns the dictionary

**Parameters** **`filename`** – name of the yaml file

**Returns** dictionary of YAML file contents

`policy_sentry.util.file.`**`write_json_file`**(*filename*, *json_contents*)

Description: Writes a YAML file :param json_contents: a dictionary used to build the JSON. This is the IAM Policy built by write_policy functions. :param filename: name of the yaml file, which should include the path

### util.actions

Text operations specific to IAM actions

`policy_sentry.util.actions.`**`get_action_name_from_action`**(*action*)

Returns the lowercase action name from a service:action combination :param action: ec2:DescribeInstance :return: describeinstance

`policy_sentry.util.actions.`**`get_full_action_name`**(*service*, *action_name*)

Gets the proper formatting for an action - the service, plus colon, plus action name. :param service: service name, like s3 :param action_name: action name, like createbucket :return: the resulting string

`policy_sentry.util.actions.`**`get_lowercase_action_list`**(*action_list*)

Given a list of actions, return the list but in lowercase format

`policy_sentry.util.actions.`**`get_service_from_action`**(*action*)

Returns the service name from a service:action combination :param action: ec2:DescribeInstance :return: ec2

# CHAPTER 13

# Implementation Strategy

In the context of your overall organization strategy for AWS IAM, we recommend using a few measures for locking down your AWS environments with IAM:

1. Use policy_sentry to create Identity-based policies
2. Use Service Control Policies (SCPs) to lock down available API calls per account.
    - A great collection of SCPs can be found on asecure.cloud.
    - Control Tower has some excellent guidance on strategy for SCPs in their documentation. Note that they call it "Guardrails" but they are mostly SCPs. See the docs here
3. Use Repokid to revoke out of date policies as your application/roles mature.
4. Use Resource-based policies for all services that support them.
    - A list of which services support resource-based policies can be found in the AWS documentation here.
5. Never provision infrastructure manually; use Infrastructure as Code
    - I highly suggest Terraform for IAC over other alternatives such as CloudFormation, Chef, or Puppet. Yevgeniy Brikman explains the reasons very well in this Gruntwork.io blog post.
    - I also suggest reading HashiCorp's Unlocking the Cloud Operating Model Whitepaper.

# CHAPTER 14

# IAM Policies

This document covers:

- Elements of an IAM Policy
- Breakdown of the tables for Actions, Resources, and Condition keys per service
- Generally how policy_sentry uses these tables to generate IAM Policies

## 14.1 IAM Policy Elements

The following IAM JSON Policy elements are included in policy_sentry-generated IAM Policies:

- Version: specifies policy language versions dictated by AWS. There are two options - `2012-10-17` and `2008-10-17`. policy_sentry generates policies for the most recent policy language - `2012-10-17`
- Statement: There is one **statement array** per policy, with multiple statements/SIDs inside that statement. The elements of a single statement/SID are listed below.
  - SID: Statement ID. Optional identifier for the policy statement. SID values can be assigned to each statement in a statement array.
  - Effect: `Allow` or explicit `Deny`. If there is any overlap on an action or actions with Allow vs. Deny, the `Deny` effect overrides the `Allow`.
  - Action: This refers to the IAM action - i.e., `s3:GetObject`, or `ec2:DescribeInstances`. Action text in a statement can have wildcards included: for example, `ec2:*` covers all EC2 actions, and `ec2:Describe*` covers all EC2 actions prefixed with `Describe` - such as `DescribeInstances`, `DescribeInstanceAttributes`, etc.
  - Resource: This refers to an Amazon Resource Name (ARN) that the Action can be performed against. There are differences in ARN format per service. Those differences can be viewed in the AWS Docs on ARNs and Namespaces

The ones we don't use in this tool:

- Condition (will be added in a future release)

- Principal
- NotPrincipal
- NotResource

## 14.2 Actions, Resources, and Condition Keys Per Service

> If you *ever* write or review IAM Policies, you should bookmark the documentation page for AWS Actions, Resources, and Context Keys here

This documentation is the seed source for the database that we create in policy_sentry. It contains tables for **(1) Actions**, **(2) Resources/ARNs**, and **(3) Condition Keys** for each service. This documentation is of critical importance because **every IAM action for every IAM service has different ARNs that it can apply to, and different Condition Keys that it can apply to.**

### 14.2.1 Action Table

Consider the Action table snippet from KMS shown below (source documentation can be viewed on the KMS documentation here).

| Actions | Access Level | Resource Types | Condition Keys | Dependent Actions |
|---|---|---|---|---|
| kms:CreateGrant | Permissions management | key* | | |
| | | • | kms:CallerAccount | |
| kms:CreateCustomKeyStore | Write | • | | cloudhsm:DescribeClusters |

As you can see, the Actions Table contains these columns:

- **Actions**: The name of the IAM Action
- **Access Level**: how the action is classified. This is limited to List, Read, Write, Permissions management, or Tagging.
    - This classification can help you understand the level of access that an action grants when you use it in a policy.
    - For more information about access levels, see Understanding Access Level Summaries Within Policy Summaries.
- **Condition Keys**: The condition key available for that action. There are some service specific ones that will contain the service namespace (i.e., `ec2`, or in this case, `kms`. Sometimes, there are AWS-level condition keys that are available to only some actions within some services, such as aws:SourceAccount. If those are available to the action, they will be supplied in that column.
- **Dependent Actions**: Some actions require that other actions can be executed by the IAM Principal. The example above indicates that in order to call `kms:CreateCustomKeyStore`, you must be able to also execute `cloudhsm:DescribeClusters`.

**And most importantly** to the context of this tool, there is the Resource Types column:

- **Resource Types**: This indicates whether the action supports resource-level permissions - i.e., *restricting IAM Actions by ARN*. If there is a value here, it points to the ARN Table shown later in the documentation.

  - In the example above, you can see that `kms:CreateCustomKeyStore`'s Resource Types cell is blank; this indicates that `kms:CreateCustomKeyStore` can **only** have `*` as the resource type.
  - Conversely, for `kms:CreateGrant`, the action can have either (1) `*` as the resource type, or `key*` as the resource type. The ARN format is not actually `key*`, it just points to that ARN format in the ARN Table explained below.

### 14.2.2 ARN Table

Consider the KMS ARN Table shown below (the source documentation can be viewed on the AWS website here.

| Resource Types | ARN | Condition Keys |
|---|---|---|
| alias | `arn:${Partition}:kms:${Region}:${Account}:alias/${Alias}` | |
| key | `arn:${Partition}:kms:${Region}:${Account}:key/${KeyId}` | |

The ARN Table has three fields:

- **Resource Types**: The name of the resource type. This corresponds to the "Resource Types" field in the Action table. In the example above, the types are:
- `alias`
- `key`
- **ARN**: This shows the required ARN format that can be specified in IAM policies for the IAM Actions that allow this ARN format. In the example above the ARN types are:
- `arn:${Partition}:kms:${Region}:${Account}:alias/${Alias}`
- `arn:${Partition}:kms:${Region}:${Account}:key/${KeyId}`
- **Condition Keys**: This specifies condition context keys that you can include in an IAM policy statement only when both (1) this resource and (2) a supporting action from the table above are included in the statement.

### 14.2.3 Condition Keys Table

There is also a Condition Keys table. An example is shown below.

| Condition Keys | Type | Description |
|---|---|---|
| `kms:BypassPolicyLockoutSafetyCheck` | Bool | Controls access to the CreateKey and PutKeyPolicy operations based on the value of the BypassPolicyLockoutSafetyCheck parameter in the request. |
| `kms:CallerAccount` | String | Controls access to specified AWS KMS operations based on the AWS account ID of the caller. You can use this condition key to allow or deny access to all IAM users and roles in an AWS account in a single policy statement. |

**Note**: While policy_sentry does import the Condition Keys table into the database, it does not currently provide functionality to insert these condition keys into the policies. This is due to the complexity of each condition key, and the dubious viability of mandating those condition keys for every IAM policy.

We might support the Global Condition keys for IAM policies in the future, perhaps to be supplied via a user config file, but that functionality is not on the roadmap at this time. For more information on Global Condition Keys, see this documentation.

### 14.2.4 References

- ARN Formats and Service Namespaces
- IAM Policy Elements
- IAM Actions, Resources, and Context Keys per service
- Actions Table explanation
- ARN Table explanation
- Condition Keys Table explanation
- Global Condition Keys

# CHAPTER 15

# Minimization

This document explains the approach in the file titled `policy_sentry/shared/minimize.py`, which is heavily borrowed from Netflix's policyuniverse

IAM Policies have character limits, which apply to individual policies, and there are also limits on the total aggregate policy sizes. As such, it is not possible to use exhaustive list of explicit IAM actions. To have granular control of specific IAM policies, we must use wildcards on IAM Actions, only in a programmatic manner.

This is typically performed by humans by reducing policies to `s3:Get*`, `ec2:Describe*`, and other approaches of the sort.

Netflix's PolicyUniverse[1] has addressed this problem using a few functions that we borrowed directly, and slightly modified. All of these functions are inside the aforementioned `minimize.py` file, and are also listed below:

- get_denied_prefixes_from_desired
- check_min_permission_length
- minimize_statement_actions

We modified the functions, in short, because of how we source our list of IAM actions. Policyuniverse leverages a file titled `data.json`, which appears to be a manually altered version of the policies.js file included as part of the AWS Policy Generator website. However, that page is not updated as frequently. It also does not include the same details that we get from the Actions, Resources, and Condition Keys page, like the Dependent Actions Field, service-specific conditions, and most importantly the multiple ARN format types that can apply to any particular IAM Action.

See the AWS IAM FAQ page for supporting details on IAM Size. For your convenience, the relevant text is clipped below.

> Q: How many policies can I attach to an IAM role?
>
> - For inline policies: You can add as many inline policies as you want to a user, role, or group, but the total aggregate policy size (the sum size of all inline policies) per entity cannot exceed the following limits:
>   - User policy size cannot exceed 2,048 characters.
>   - Role policy size cannot exceed 10,240 characters.
>   - Group policy size cannot exceed 5,120 characters.

- For managed policies: You can add up to 10 managed policies to a user, role, or group.
- The size of each managed policy cannot exceed 6,144 characters.

# CHAPTER 16

## Indices and tables

- modindex

# Python Module Index

p

`policy_sentry.analysis.analyze`, 69
`policy_sentry.command.write_policy`, 65
`policy_sentry.querying.actions`, 60
`policy_sentry.querying.all`, 60
`policy_sentry.querying.arns`, 63
`policy_sentry.querying.conditions`, 64
`policy_sentry.util.actions`, 73
`policy_sentry.util.arns`, 72
`policy_sentry.util.file`, 72
`policy_sentry.util.policy_files`, 71
`policy_sentry.writing.minimize`, 68
`policy_sentry.writing.sid_group`, 65
`policy_sentry.writing.template`, 68
`policy_sentry.writing.validate`, 68

# Index

## A

`add_action_without_resource_constraint()` (*policy_sentry.writing.sid_group.SidGroup method*), 66
`add_by_arn_and_access_level()` (*policy_sentry.writing.sid_group.SidGroup method*), 66
`add_by_list_of_actions()` (*policy_sentry.writing.sid_group.SidGroup method*), 66
`analyze_by_access_level()` (*in module policy_sentry.analysis.analyze*), 69
`analyze_policy_directory()` (*in module policy_sentry.analysis.analyze*), 69
`analyze_policy_file()` (*in module policy_sentry.analysis.analyze*), 70
`analyze_statement_by_access_level()` (*in module policy_sentry.analysis.analyze*), 70
`arn_has_colons()` (*in module policy_sentry.util.arns*), 72
`arn_has_slash()` (*in module policy_sentry.util.arns*), 72

## C

`check()` (*in module policy_sentry.writing.validate*), 68
`check_actions_schema()` (*in module policy_sentry.writing.validate*), 68
`check_crud_schema()` (*in module policy_sentry.writing.validate*), 68
`check_min_permission_length()` (*in module policy_sentry.writing.minimize*), 69
`check_valid_file_path()` (*in module policy_sentry.util.file*), 72
`create_actions_template()` (*in module policy_sentry.writing.template*), 68
`create_crud_template()` (*in module policy_sentry.writing.template*), 68
`create_directory_if_it_doesnt_exist()` (*in module policy_sentry.util.file*), 73
`create_policy_sid_namespace()` (*in module policy_sentry.writing.sid_group*), 67

## D

`determine_actions_to_expand()` (*in module policy_sentry.analysis.analyze*), 70
`determine_risky_actions()` (*in module policy_sentry.analysis.analyze*), 71
`determine_risky_actions_from_list()` (*in module policy_sentry.analysis.analyze*), 71
`does_arn_match()` (*in module policy_sentry.util.arns*), 72

## E

`expand()` (*in module policy_sentry.analysis.analyze*), 71

## G

`get_account_from_arn()` (*in module policy_sentry.util.arns*), 72
`get_action_data()` (*in module policy_sentry.querying.actions*), 60
`get_action_name_from_action()` (*in module policy_sentry.util.actions*), 73
`get_actions_at_access_level_that_support_wildcard_a` (*in module policy_sentry.querying.actions*), 61
`get_actions_for_service()` (*in module policy_sentry.querying.actions*), 61
`get_actions_from_json_policy_file()` (*in module policy_sentry.util.policy_files*), 71
`get_actions_from_policy()` (*in module policy_sentry.util.policy_files*), 71
`get_actions_from_statement()` (*in module policy_sentry.util.policy_files*), 71
`get_actions_matching_condition_crud_and_arn()` (*in module policy_sentry.querying.actions*), 61
`get_actions_matching_condition_key()` (*in module policy_sentry.querying.actions*), 61
`get_actions_template_dict()` (*in module policy_sentry.writing.template*), 68

get_actions_that_support_wildcard_arns_only() (*in module policy_sentry.querying.actions*), 62
get_actions_with_access_level() (*in module policy_sentry.querying.actions*), 62
get_actions_with_arn_type_and_access_level() (*in module policy_sentry.querying.actions*), 62
get_all_actions() (*in module policy_sentry.querying.all*), 60
get_all_service_prefixes() (*in module policy_sentry.querying.all*), 60
get_arn_data() (*in module policy_sentry.querying.arns*), 63
get_arn_type_details() (*in module policy_sentry.querying.arns*), 63
get_arn_types_for_service() (*in module policy_sentry.querying.arns*), 64
get_condition_key_details() (*in module policy_sentry.querying.conditions*), 64
get_condition_keys_available_to_raw_arn() (*in module policy_sentry.querying.conditions*), 64
get_condition_keys_for_service() (*in module policy_sentry.querying.conditions*), 65
get_condition_value_type() (*in module policy_sentry.querying.conditions*), 65
get_conditions_for_action_and_raw_arn() (*in module policy_sentry.querying.conditions*), 65
get_crud_template_dict() (*in module policy_sentry.writing.template*), 68
get_denied_prefixes_from_desired() (*in module policy_sentry.writing.minimize*), 69
get_dependent_actions() (*in module policy_sentry.querying.actions*), 62
get_full_action_name() (*in module policy_sentry.util.actions*), 73
get_lowercase_action_list() (*in module policy_sentry.util.actions*), 73
get_partition_from_arn() (*in module policy_sentry.util.arns*), 72
get_raw_arns_for_service() (*in module policy_sentry.querying.arns*), 64
get_region_from_arn() (*in module policy_sentry.util.arns*), 72
get_rendered_policy() (*policy_sentry.writing.sid_group.SidGroup method*), 66
get_resource_from_arn() (*in module policy_sentry.util.arns*), 72
get_resource_path_from_arn() (*in module policy_sentry.util.arns*), 72
get_resource_type_name_with_raw_arn() (*in module policy_sentry.querying.arns*), 64
get_service_from_action() (*in module policy_sentry.util.actions*), 73
get_service_from_arn() (*in module policy_sentry.util.arns*), 72
get_sid() (*policy_sentry.writing.sid_group.SidGroup method*), 66
get_sid_group() (*policy_sentry.writing.sid_group.SidGroup method*), 67
get_universal_conditions() (*policy_sentry.writing.sid_group.SidGroup method*), 67

L

list_files_in_directory() (*in module policy_sentry.util.file*), 73
list_sids() (*policy_sentry.writing.sid_group.SidGroup method*), 67

M

minimize_statement_actions() (*in module policy_sentry.writing.minimize*), 69

P

parse_arn() (*in module policy_sentry.util.arns*), 72
policy_sentry.analysis.analyze (*module*), 69
policy_sentry.command.write_policy (*module*), 65
policy_sentry.querying.actions (*module*), 60
policy_sentry.querying.all (*module*), 60
policy_sentry.querying.arns (*module*), 63
policy_sentry.querying.conditions (*module*), 64
policy_sentry.util.actions (*module*), 73
policy_sentry.util.arns (*module*), 72
policy_sentry.util.file (*module*), 72
policy_sentry.util.policy_files (*module*), 71
policy_sentry.writing.minimize (*module*), 68
policy_sentry.writing.sid_group (*module*), 65
policy_sentry.writing.template (*module*), 68
policy_sentry.writing.validate (*module*), 68
process_template() (*policy_sentry.writing.sid_group.SidGroup method*), 67

R

read_risky_iam_permissions_text_file() (*in module policy_sentry.analysis.analyze*), 71 `read_this_file()` (*in module policy_sentry.util.file*), 73

`read_yaml_file()` (*in module policy_sentry.util.file*), 73

`remove_actions_duplicated_in_wildcard_arn()` (*policy_sentry.writing.sid_group.SidGroup method*), 67

`remove_actions_not_matching_access_level()` (*in module policy_sentry.querying.actions*), 63

`remove_actions_not_matching_these()` (*policy_sentry.writing.sid_group.SidGroup method*), 67

`remove_actions_that_are_not_wildcard_arn_only()` (*in module policy_sentry.querying.actions*), 63

`remove_actions_that_are_not_wildcard_arn_only()` (*in module policy_sentry.writing.sid_group*), 67

## S

`SidGroup` (*class in policy_sentry.writing.sid_group*), 65

## V

`validate_condition_block()` (*in module policy_sentry.writing.validate*), 68

## W

`write_json_file()` (*in module policy_sentry.util.file*), 73

`write_policy_with_template()` (*in module policy_sentry.command.write_policy*), 65