policy$_{s}entry_{r}eadthedocs_{t}est$

Nov 12, 2019

Introduction

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:

- Perform auditing of IAM Policies based on access levels
- Query the policy database to reduce manual search time
- Download live policies from an AWS account auditing purposes
- Generate IAM Policies based on Terraform output

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.

1.1.1 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 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:

```yaml
roles_with_crud_levels:
- name: myRole
  description: '' # Insert description
  arn: '' # Insert the ARN of the role that will use this
  read:
    - '' # Insert ARNs for Read access
  write:
    - '' # Insert ARNs...
  list:
    - '' # Insert ARNs...
  tag:
    - '' # Insert ARNs...
  permissions-management:
    - '' # Insert ARNs...
```

Then just fill it out:

```yaml
roles_with_crud_levels:
- name: myRole
  description: 'Justification for privileges'
  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'
  tag:
    - 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
```

(continues on next page)

(continued from previous page)

```
  permissions-management:
    - 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
```

Then run this command:

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

(continues on next page)

(continued from previous page)

```
            "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.

1.2 Installation

- `policy_sentry` is available via pip. To install, run:

```
pip install --user policy_sentry
```

1.2.1 Usage

- `initialize`: Create a SQLite database that contains all of the services available through the Actions, Resources, and Condition Keys documentation. See the documentation.
- `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.
- `download-policies`: Download IAM policies from your AWS account for analysis.
- *analyze-iam-policy*: Analyze an IAM policy read from a JSON file, expands the wildcards (like *s3:List** if necessary.
  - Option 1: Audits them to see if certain IAM actions are permitted, based on actions in a separate text file. See the documentation on Initialization.
  - Option 2: Audits them to see if any of the actions in the policy meet a certain access level, such as "Permissions management."

1.3 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

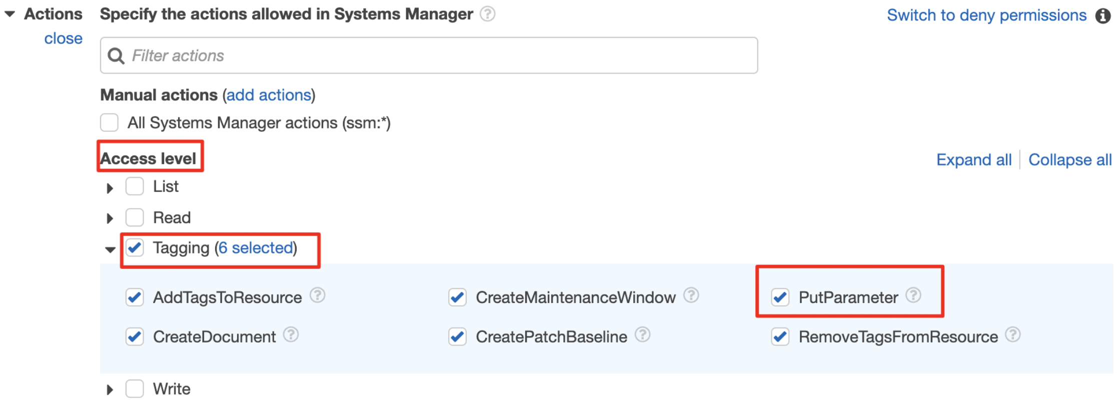
- AWS IAM Visual Policy Editor in the AWS Console

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.



▾ Actions
close

Specify the actions allowed in Systems Manager

Switch to deny permissions

Filter actions

Manual actions (add actions)

□ All Systems Manager actions (ssm:*)

Access level

Expand all | Collapse all

▸ □ 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 --crud`, 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, we still need to make those policies secure by default.

CHAPTER 3

Installation

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

```
pip install --user policy_sentry
```

CHAPTER 4

Initialization

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

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.

4.1 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.

4.2 Usage

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

# Initialize the database with a custom Access Level Overrides file
policy_sentry initialize --access-level-overrides-file my-override.yml
```

CHAPTER 5

Writing IAM Policies

5.1 CRUD Mode: ARNs 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

- `--crud`: Specify this option to use the CRUD functionality. File must be formatted as expected. Defaults to false.
- `--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`.

Example:

```
policy_sentry write-policy --crud --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:

```
roles_with_crud_levels:
- name: myRole
  description: '' # Insert description
  arn: '' # Insert the ARN of the role that will use this
  read:
    - '' # Insert ARNs for Read access
  write:
    - '' # Insert ARNs...
  list:
    - '' # Insert ARNs...
  tag:
    - '' # Insert ARNs...
  permissions-management:
    - '' # Insert ARNs...
```

- Then just fill it out:

```
roles_with_crud_levels:
- name: myRole
  description: 'Justification for privileges'
  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'
  tag:
    - '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 --crud --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": [
                "arn:aws:ssm:us-east-1:123456789012:parameter/myparameter"
            ]
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "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: Lists of IAM Actions

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

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 tmp.yml --template-type actions
```

- It will generate a file with contents like this:

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

- Create a yaml file with the following contents:

```
roles_with_actions:
- name: 'RoleNameWithActions'
  description: 'Justification for privileges' # for auditability
  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 examples/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/${KeyId}"
            ]
        },
        {
            "Sid": "Ec2WriteSecuritygroup",
            "Effect": "Allow",
            "Action": [
                "ec2:authorizesecuritygroupegress",
                "ec2:authorizesecuritygroupingress"
            ],
            "Resource": [
                "arn:aws:ec2:${Region}:${Account}:security-group/${SecurityGroupId}"
            ]
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "Sid": "MultMultNone",
            "Effect": "Allow",
            "Action": [
                "kms:createcustomkeystore",
                "cloudhsm:describeclusters"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

5.3 Folder Mode: Write Multiple Policies from CRUD mode files

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]
  --crud              Use the CRUD functionality. Defaults to false
  --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.
```

CHAPTER 6

Downloading Policies

```
Usage: policy_sentry download-policies [OPTIONS]

  Download remote IAM policies to a directory for use in the analyze-iam-
  policies command.

Options:
  --profile TEXT         To authenticate to AWS and analyze *all* existing IAM␣
↪policies.
  --aws-managed          Use flag if you want to download AWS Managed policies too.
  --include-unattached   Download policies that are unattached too. Defaults to false.
  --help                 Show this message and exit.
```

- Make sure you are authenticated to AWS.

6.1 Customer-managed policies

- Run this command:

```
policy_sentry download-policies --profile dev
```

- It will download the policies to `$HOME/.policy_sentry/policy-analysis/account-number/customer-managed`.
- You can then run analysis on the entire directory:

```
policy_sentry analyze-iam-policy --policy $HOME/.policy_sentry/policy-analysis/
↪0123456789012/customer-managed --from-access-level permissions-management
```

Then it will print out the IAM policies that contain actions with “Permissions management” access levels.

6.2 AWS Managed policies

- Run this command:

```
policy_sentry download-policies --profile dev --aws-managed
```

- It will download the policies to `$HOME/.policy_sentry/policy-analysis/account-number/aws-managed`.
- You can then run analysis on the entire directory:

```
analyze-iam-policy --policy $HOME/.policy_sentry/policy-analysis/0123456789012/
↪customer-managed --from-access-level permissions-management
```

Then it will print out the AWS Managed IAM policies that contain actions with “Permissions management” access levels.

CHAPTER 7

Analyzing Policies

`analyze-iam-policy`: Reads a policy from a JSON file, expands the wildcards (like `s3:List*` if necessary, and audits them to see if certain IAM actions are permitted.

7.1 Motivation

Let's say you are a developer that handles creation of IAM policies.

- An internal customer asks you to create an IAM policy.
- You haven't been tasked with auditing IAM policies yourself, as that's not your area of expertise, and until this point there is no automation to do it for you.
- However, you want to make sure that the customers aren't asking for permissions that they don't need, since we need to have *some* guardrails in place to prevent unnecessary exposure of attack surfaces.
- This is made more difficult by the fact that sometimes, the customer will give you IAM policies that include `*` in the actions. Not only do you want to restrict actions to the specific ARNs, but you want to know what actions they actually need!

You can solve this with policy_sentry too, by auditing for IAM actions in a given policy. Tell them to supply the policy to you in JSON format, and feed it into the `analyze_iam_policy` command, as shown below.

7.2 Options

```
Usage: policy_sentry analyze-iam-policy [OPTIONS]

  Analyze IAM Actions given a JSON policy file

Options:
  --from-audit-file TEXT          The file containing AWS actions to audit. Default␣
↪path is $HOME/.policy_sentry/audit/permissions-access-level.txt.
```

(continues on next page)

(continued from previous page)

```
  --from-access-level [read|write|list|tagging|permissions-management]
                                  Show CRUD levels. Acceptable values are read, write,
↪ list, tagging, permissions-management
  --policy TEXT                   Supply the requester's IAM policy as a JSON file.␣
↪Accepts relative path.  [required]
  --help                          Show this message and exit.
```

7.3 Instructions

- Build the database:

```
policy_sentry initialize
```

7.3.1 Audit for custom list of actions

- You can specify your own audit file.

```
policy_sentry analyze-iam-policy --from-audit-file ~/.policy_sentry/audit/privilege-
↪escalation.txt --policy examples/analyze/wildcards.json
```

- `policy_sentry` comes bundled with two different audit files, which are located in the `~/.policy_sentry/audit` directory.

  1. privilege-escalation.txt: This is based off of Rhino Security Labs research
  2. resource-exposure.txt: This is a list of all "Permissions management" actions from the `policy_sentry` database.

We plan on supporting more pre-bundled audit files in the future

7.3.2 Audit a policy file for permissions with specific access levels

- Command:

```
policy_sentry analyze-iam-policy --from-access-level permissions-management --policy␣
↪examples/analyze/wildcards.json
```

- Output:

```
Evaluating: examples/analyze/wildcards.json
Access level: permissions-management
[   'ecr:setrepositorypolicy',
    's3:objectowneroverridetobucketowner',
    's3:putbucketpolicy',
    's3:putbucketacl',
    's3:putobjectacl',
    's3:putobjectversionacl',
    's3:putaccountpublicaccessblock',
    's3:putbucketpublicaccessblock',
    's3:deletebucketpolicy']
```

7.3.3 Audit entire folders

- `policy_sentry` will detect folders vs. files automatically. Just run the command as usual:

```
policy_sentry analyze-iam-policy --from-access-level permissions-management --policy /
↪Users/username/.policy_sentry/policy-analysis/0123456789012/aws-managed
```

```
Evaluating policy files in /Users/username/.policy_sentry/policy-analysis/
↪0123456789012/aws-managed

Policy: AmazonSageMakerFullAccess.json
[   'ec2:createnetworkinterfacepermission',
    'ec2:deletenetworkinterfacepermission',
    'ecr:setrepositorypolicy',
    'iam:createservicelinkedrole',
    'iam:createservicelinkedrole',
    'iam:passrole']

Policy: AmazonEC2RoleforDataPipelineRole.json
[   's3:putobjectacl',
    's3:putbucketpolicy',
    's3:putbucketacl',
    's3:objectowneroverridetobucketowner',
    's3:putobjectversionacl',
    's3:putbucketpublicaccessblock',
    's3:deletebucketpolicy',
    's3:putaccountpublicaccessblock',
    'sns:removepermission',
    'sns:addpermission',
    'sqs:addpermission',
    'sqs:removepermission']

Policy: AWSDataPipelineRole.json
[   'elasticmapreduce:putblockpublicaccessconfiguration',
    'iam:createservicelinkedrole',
    'iam:passrole',
    's3:putobjectacl',
    's3:putbucketpolicy',
    's3:putbucketacl',
    's3:putobjectversionacl',
    's3:putbucketpublicaccessblock',
    's3:putaccountpublicaccessblock']

Policy: AWSSupportServiceRolePolicy.json
['iam:deleterole', 'lightsail:getinstanceaccessdetails']
```

CHAPTER 8

Querying the 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.

8.1 Commands

- Query the **Action** table:

```
# Get a list of all IAM Actions available to the RAM service
policy_sentry query --table action --service ram
# Get details about the `ram:TagResource` IAM Action
policy_sentry query --table action --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 --table action --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 --table action --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 --table arn --service ssm
# Get the raw ARN format for the `cloud9` ARN with the short name `environment`
policy_sentry query --table arn --service cloud9 --name environment
```

- Query the **Condition Keys** table:

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

CHAPTER 9

Command cheat sheet

9.1 Commands

- initialize: Create a SQLite database that contains all of the services available through the Actions, Resources, and Condition Keys documentation. See the [documentation][12].
- download-policies: Download IAM policies from an AWS account locally for analysis against the database.
- 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.
- analyze-iam-policy: Analyze an IAM policy read from a JSON file, expands the wildcards (like s3:List* if necessary.
  - Option 1: Audits them to see if certain IAM actions are permitted, based on actions in a separate text file. See the [documentation][12].
  - Option 2: Audits them to see if any of the actions in the policy meet a certain access level, such as “Permissions management.”

9.2 Policy Writing Commands

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

(continues on next page)

(continued from previous page)

```
# Create a template file for use in the write-policy command (crud mode)
policy_sentry create-template --name myRole --output-file tmp.yml --template-type crud

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

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

# Create a template file for use in the write-policy command (actions mode)
policy_sentry create-template --name myRole --output-file tmp.yml --template-type␣
↪actions

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

9.3 Policy Analysis Commands

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

# Analyze a policy FILE to determine actions with "Permissions Management" access␣
↪levels
policy_sentry analyze-iam-policy --from-access-level permissions-management --file␣
↪examples/analyze/wildcards.json

# Download customer managed IAM policies from a live account under 'default' profile.␣
↪By default, it looks for policies that are 1. in use and 2. customer managed
policy_sentry download-policies # this will download to ~/.policy_sentry/accountid/
↪customer-managed/.json

# Download customer-managed IAM policies, including those that are not attached
policy_sentry download-policies --include-unattached # this will download to ~/.
↪policy_sentry/accountid/customer-managed/.json

# Analyze a DIRECTORY of policy files
policy_sentry analyze-iam-policy --show ~/.policy_sentry/123456789012/customer-managed

# Analyze a policy FILE to identify higher-risk IAM calls
policy_sentry analyze-iam-policy --file examples/analyze/wildcards.json

# Analyze a policy against a custom file containing a list of IAM actions
policy_sentry analyze-iam-policy --file examples/analyze/wildcards.json --from-audit-
↪file ~/.policy_sentry/audit/privilege-escalation.txt
```

CHAPTER 10

Terraform Demo

Please download the demo code here to follow along.

10.1 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]
  --crud              Use the CRUD functionality. Defaults to false
  --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.
```

10.2 Prerequisites

This requires:

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

10.3 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 --crud --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 11

Terraform Modules

11.1 1: Install policy_sentry

- Install policy_sentry

```
pip3 install policy_sentry
```

- Initialize policy_sentry

```
policy_sentry initialize
```

11.2 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.

11.3 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 --crud --input-dir files --output-dir files
```

11.4 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 12

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

12.1 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

12.2 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.**

12.2.1 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.

12.2.2 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.

12.2.3 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.

12.2.4 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 13

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 14

Contributing

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

14.1 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 below. 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.)

14.2 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
```

14.2.1 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.

14.3 Coding Standards

- Use Python Black to adhere to pep8 automagically.

14.4 Developing Against HEAD

14.4.1 Pipenv

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

14.5 Running the Test Suite

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

- Just run the following:

```
nosetests -v
```

Output:

```
Tests the format of the overrides yml file for the RAM service ... ok
Tests iam:CreateAccessKey (in overrides file as Permissions management, but in the␣
↪AWS docs as Write) ... ok
test_get_actions_by_access_level (test_actions.ActionsTestCase) ... ok
test_get_dependent_actions_double (test_actions.ActionsTestCase) ... ok
```

(continues on next page)

(continued from previous page)

```
test_get_dependent_actions_several (test_actions.ActionsTestCase) ... ok
test_get_dependent_actions_single (test_actions.ActionsTestCase) ... 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_actions_template (test_template.TemplateTestCase) ... ok
test_crud_template (test_template.TemplateTestCase) ... ok
test_print_policy_with_actions_having_dependencies (test_write_policy.
↪WritePolicyActionsTestCase) ... ok
test_write_policy (test_write_policy.WritePolicyCrudTestCase) ... 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 --crud when the YAML␣
↪file is missing access level categories ... 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
```

14.6 Updating the AWS HTML files

Run the following:

```
./utils/grab-docs.sh
# Or:
./utils/download-docs.sh
```

CHAPTER 15

Contribution Guidelines

Fill this in later.

CHAPTER 16

Internals

Before reading this, make sure you have read all the other documentation - especially the IAM Policies document, which covers the Action Tables, ARN tables, and Condition Keys Tables.

Other assumptions:

- You are familiar with these Python things:
  - click
  - package imports, multi folder management
  - PyPi
  - Unit testing

16.1 Overall: How policy_sentry uses these tables

policy_sentry follows this process for generating policies.

1. If **User-supplied actions** is chosen:
   - 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** (i.e., the `--crud` flag) is chosen:
   - 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** is chosen:
   - Associate the IAM actions requested by the user to the service, access level, and ARN type matching the aforementioned SID namespace
6. If **User-supplied ARNs with Access levels** (i.e., the `--crud` flag) is chosen:
   - Associate all the IAM actions that correspond to the service, access level, and ARN type matching that SID namespace.
7. Print the policy

16.2 Project Structure

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

16.2.1 Subfolders

- `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.
- `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.
- *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*.

16.2.2 Files

TODO: Insert brief explanations of strategy for some of these later. It will just help people as they try to figure out this project.

- actions.py
- analyze.py
- arns.py
- conditions.py
- config.py
- database.py
- download.py
- file.py
- links.yml

- login.py
- minimize.py
- roles.py
- scrape.py
- template.py

CHAPTER 17

Roadmap

- 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.

17.1 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 18

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 19

Policy Sentry as a Python Package

TODO: Instructions for using it as a python package so you can query the database.