PolicyGlass

Release 0.8.0

Sam Martin

Jan 09, 2022

INDEX

PolicyGlass is an effective permission parser for AWS Policies. It takes normal JSON policies of any type (Principal, Resource, or Endpoint) and converts them into `PolicyShard` objects that are *always* assertions about what is allowed.

CHAPTER

ONE

USE CASES

There are two main use cases for this tool:

1. Writing tools that audit the permissions provided to AWS resources/principals
2. Validating your understanding of the complex policy you're writing.

CHAPTER

TWO

WHY DO I NEED POLICYGLASS?

Isn't this a simple problem? I can just check actions and resources in each statement, boom, done.

Understanding AWS policies programmatically is harder than it looks.

You can write code easily enough to check what resources and actions are in each statement, and that might seem like enough. But what happens when you throw a `Deny` statement into the mix? Well that's okay, you just check each statement to see if it's an allow or a deny and if it's a deny then you just remove any resources from the allow that exist in the deny right? Easy enough, but what about resources that are just `*` or are ARNs with wildcards in them? Once you've got past that, you have to deal with statements that contain negations (`NotAction`, `NotResource`, and `NotPrincipal`), it's starting to get harder. Then you have to add in the complexity of conditions, and all this is without even mentioning the complexity of parsing an AWS Policy in the first place with the variants of `Actions` as a list or as a string, or `Resources` that may be a string or a dictionary.

PolicyGlass takes care of all this for you by breaking down a policy into its components and applying set operations in order to build shards that describe the effective permissions.

CHAPTER

THREE

EXAMPLES OF POLICYSHARDS

Below you can find some examples on how PolicyGlass can be used to understand complex policies in a consistent way.

We're going to use *policy_shards_to_json()* to make the output a bit easier to read.

Tip: Remember *PolicyShard* objects are *not* policies. They represent policies in an abstracted way that makes them easier to understand programmatically, the JSON output you see in the examples is not a policy you can use directly in AWS.

3.1 Simple

```
>>> from policyglass import Policy, dedupe_policy_shards, policy_shards_effect, policy_
↪shards_to_json
>>> policy_a = Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Allow",
...             "Action": [
...                 "s3:*"
...             ],
...             "Resource": "*"
...         }
...     ]
... })
>>> policy_b = Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Deny",
...             "Action": [
...                 "s3:*"
...             ],
...             "Resource": "arn:aws:s3:::examplebucket/*"
...         }
...     ]
... })
```

(continues on next page)

(continued from previous page)

```
>>> policy_shards = policy_shards_effect([*policy_a.policy_shards, *policy_b.policy_
↪shards])
>>> print(policy_shards_to_json(policy_shards, exclude_defaults=True, indent=2))
[
   {
     "effective_action": {
       "inclusion": "s3:*"
     },
     "effective_resource": {
       "inclusion": "*",
       "exclusions": [
         "arn:aws:s3:::examplebucket/*"
       ]
     },
     "effective_principal": {
       "inclusion": {
         "type": "AWS",
         "value": "*"
       }
     }
   }
 ]
```

PolicyShard #1 (first dictonary in list) tells us:

1. *s3:** is allowed for all resources **except** `arn:aws:s3:::examplebucket/*`

What occurred:

1. The `resource` from the deny was added to the allow's `EffectiveResource`'s `exclusions`

3.2 De-duplicate

```
>>> from policyglass import Policy, dedupe_policy_shards, policy_shards_to_json
>>> policy_a = Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Allow",
...             "Action": [
...                 "s3:*"
...             ],
...             "Resource": "*"
...         }
...     ]
... })
>>> policy_b = Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Allow",
...             "Action": [
```

(continues on next page)

(continued from previous page)

```
...                 "s3:*"
...             ],
...             "Resource": "*"
...         }
...     ]
... })
>>> policy_shards = dedupe_policy_shards([*policy_a.policy_shards, *policy_b.policy_
↪shards])
>>> print(policy_shards_to_json(policy_shards, exclude_defaults=True, indent=2))
[
  {
    "effective_action": {
      "inclusion": "s3:*"
    },
    "effective_resource": {
      "inclusion": "*"
    },
    "effective_principal": {
      "inclusion": {
        "type": "AWS",
        "value": "*"
      }
    }
  }
]
```

PolicyShard #1 (first dictonary in list) tells us:

1. `s3:*` is allowed on all resources.

What occurred:

1. One of the two `s3:*` policy shards was removed because it was a duplicate.

3.3 Deny Not Resource Policy

```
>>> from policyglass import Policy, policy_shards_effect, policy_shards_to_json
>>> policy_a = Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Allow",
...             "Action": [
...                 "s3:*",
...                 "s3:GetObject"
...             ],
...             "Resource": "*"
...         },
...         {
...             "Effect": "Deny",
...             "Action": [
...                 "s3:*",
```

(continues on next page)

(continued from previous page)

```
...             ],
...             "NotResource": "arn:aws:s3:::examplebucket/*",
...             "Condition": {
...                 "StringNotEquals": {
...                     "s3:x-amz-server-side-encryption": "AES256"
...                 }
...             }
...         }
...     ]
... })
>>> shards_effect = policy_shards_effect(policy_a.policy_shards)
>>> print(policy_shards_to_json(shards_effect, exclude_defaults=True, indent=2))
[
  {
    "effective_action": {
      "inclusion": "s3:*"
    },
    "effective_resource": {
      "inclusion": "arn:aws:s3:::examplebucket/*"
    },
    "effective_principal": {
      "inclusion": {
        "type": "AWS",
        "value": "*"
      }
    }
  },
  {
    "effective_action": {
      "inclusion": "s3:*"
    },
    "effective_resource": {
      "inclusion": "*",
      "exclusions": [
        "arn:aws:s3:::examplebucket/*"
      ]
    },
    "effective_principal": {
      "inclusion": {
        "type": "AWS",
        "value": "*"
      }
    },
    "effective_condition": {
      "inclusions": [
        {
          "key": "s3:x-amz-server-side-encryption",
          "operator": "StringEquals",
          "values": [
            "AES256"
          ]
        }
```

(continues on next page)

(continued from previous page)

```
        ]
      }
    }
]
```

The output has two policy shards.

PolicyShard #1 (first dictionary in list) tells us:

1. Allow `s3:*`
2. On `arn:aws:s3:::examplebucket/*`
3. No conditions

PolicyShard #2 (second dictionary in list) tells us:

1. Allow `s3:*`
2. On all resources
3. If the condition applies.

What occurred:

1. `s3:GetObject` was removed from the allow because it was totally within `s3:*`
2. A new `PolicyShard` was created with `s3:*`
3. The deny's `condition` got reversed from `StringNotEquals` to `StringEquals` and added to the new allow `PolicyShard`.

CHAPTER

FOUR

EXAMPLES OF POLICY ANALYSIS

4.1 Example Policy

Let's use a complex IAM policy as our example to demonstrate the value in analyzing policies with PolicyGlass.

```
>>> from policyglass import Policy
>>> test_policy = Policy(**{
...   "Version": "2012-10-17",
...   "Statement": [
...     {
...       "Effect": "Allow",
...       "Action": [
...         "s3:*"
...       ],
...       "Resource": "*",
...       "Condition": {
...         "NumericLessThan": {
...             "s3:TlsVersion": 1.2
...         }
...       }
...     },
...     {
...       "Effect": "Allow",
...       "Action": [
...         "s3:*"
...       ],
...       "Resource": "arn:aws:s3:::examplebucket/*"
...     },
...     {
...       "Effect": "Deny",
...       "Action": [
...         "s3:PutObject"
...       ],
...       "NotResource": "arn:aws:s3:::examplebucket/*",
...       "Condition": {
...         "StringNotEquals": {
...           "s3:x-amz-server-side-encryption": "AES256"
...         }
...       }
...     }
```

(continues on next page)

(continued from previous page)

```
...   ]
... })
```

4.1.1 Understanding a Policy

To understand the policy, let's get the *policy_shards_effect()* then use the *explain_policy_shards()* method to explain them.

```
>>> from policyglass import policy_shards_effect, explain_policy_shards
>>> test_policy_shards = policy_shards_effect(test_policy.policy_shards)
>>> explain_policy_shards(test_policy_shards)
["Allow action s3:PutObject on resource * (except for arn:aws:s3:::examplebucket/*) with␣
↪principal AWS *.
    Provided conditions s3:TlsVersion NumericLessThan ['1.2'] and s3:x-amz-server-side-
↪encryption StringEquals ['AES256'] are met.",
"Allow action s3:* (except for s3:PutObject) on resource * (except for␣
↪arn:aws:s3:::examplebucket/*) with principal AWS *.
    Provided conditions s3:TlsVersion NumericLessThan ['1.2'] are met.",
'Allow action s3:* on resource arn:aws:s3:::examplebucket/* with principal AWS *.']
```

That helps clarify what the policy results in for humans. But what if we want to programatically ask a question about what this allows?

4.1.2 Interrogating a Policy

Question: Is `s3:PutObject` allowed on `arn:aws:s3:::some-other-bucket/*`?

To answer this we need to check 2 things:

1. Is `s3:PutObject` allowed on the shard?
2. If so, is `resource arn:aws:s3:::examplebucket/*` allowed on the same shard?

As we have multiple (3) shards we have to make sure both of the answers are true for the same shard.

We can do this with a list comprehension and utilise the `in` operator to check that the *EffectiveAction* contains `s3:PutObject` and that the *EffectiveResource* contains `arn:aws:s3:::some-other-bucket/*`.

```
>>> from policyglass import Action, Resource
>>> action = Action('s3:PutObject')
>>> resource = Resource('arn:aws:s3:::some-other-bucket/*')
>>> result = [
...     shard
...     for shard in test_policy_shards
...     if action in shard.effective_action
...     and resource in shard.effective_resource
... ]
>>> result
[PolicyShard(effect='Allow',
    effective_action=EffectiveAction(inclusion=Action('s3:PutObject'),␣
↪exclusions=frozenset()),
    effective_resource=EffectiveResource(inclusion=Resource('*'), exclusions=frozenset(
↪{Resource('arn:aws:s3:::examplebucket/*')})),
```

(continues on next page)

(continued from previous page)

```
    effective_principal=EffectivePrincipal(inclusion=Principal(type='AWS', value='*'),␣
↪exclusions=frozenset()),
    effective_condition=EffectiveCondition(inclusions=frozenset({Condition(key='s3:x-amz-
↪server-side-encryption', operator='StringEquals', values=['AES256']),
            Condition(key='s3:TlsVersion', operator='NumericLessThan', values=['1.2'])}),
        exclusions=frozenset()))]
```

From this check we can see that it is allowed by at least one shard! **But** there are two conditions.

4.1.3 Checking if Conditions exist

Whether we want to check these conditions depends on what kind of question we want to ask. Either way it's trivial to check if a condition exists or not.

```
>>> bool(result[0].effective_condition)
True
```

CHAPTER

FIVE

CLASS REFERENCE

5.1 Policy

Core Policy class.

class Policy(*, *Version=None*, *Statement*)

Main policy class.

Example

Create a policy from a dictionary.

```
>>> from policyglass import Policy
>>> Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Allow",
...             "Action": [
...                 "s3:*"
...             ],
...             "Resource": "*"
...         }
...     ]
... })
Policy(version='2012-10-17',
    statement=[Statement(effect='Allow',
        action=[Action('s3:*')],
        not_action=None,
        resource=[Resource('*')],
        not_resource=None, principal=None,
        not_principal=None,
        condition=None)])
```

Parameters

- **Version** (*str*) –
- **Statement** (*List[policyglass.statement.Statement]*) –

Return type None

policy_json()
Return a valid policy JSON from this policy.

Return type str

property policy_shards: List[*policyglass.policy_shard.PolicyShard*]
Shatter this policy into a number *policyglass.policy_shard* objects.

statement: List[*policyglass.statement.Statement*]

version: Optional[str]

5.2 Policy Shard

PolicyShards are a simplified representation of policies.

class PolicyShard(*effect*, *effective_action*, *effective_resource*, *effective_principal*, *effective_condition=None*)
A PolicyShard is part of a policy broken down in such a way that it can be deduplicated and collapsed.

Parameters

- **effect** (*str*) –
- **effective_action** (*policyglass.effective_arp.EffectiveARP[policyglass.action.Action]*) –
- **effective_resource** (*policyglass.effective_arp.EffectiveARP[policyglass.resource.Resource]*) –
- **effective_principal** (*policyglass.effective_arp.EffectiveARP[policyglass.principal.Principal]*) –
- **effective_condition** (policyglass.condition.EffectiveCondition) –

Return type None

class Config
Pydantic Config.

json_encoders = {<class 'policyglass.action.EffectiveAction'>: <function PolicyShard.Config.<lambda>>, <class 'policyglass.resource.EffectiveResource'>: <function PolicyShard.Config.<lambda>>, <class 'policyglass.principal.EffectivePrincipal'>: <function PolicyShard.Config.<lambda>>}

__init__(*effect*, *effective_action*, *effective_resource*, *effective_principal*, *effective_condition=None*)
Initialize a PolicyShard object.

Parameters

- **effect** (*str*) – *'Allow'* or *'Deny'*
- **effective_action** (*policyglass.effective_arp.EffectiveARP[policyglass.action.Action]*) – The EffectiveAction that this PolicyShard allows or denies
- **effective_resource** (*policyglass.effective_arp.EffectiveARP[policyglass.resource.Resource]*) – The EffectiveResource that this PolicyShard allows or denies
- **effective_principal** (*policyglass.effective_arp.EffectiveARP[policyglass.principal.Principal]*) – The EffectivePrincipal that this PolicyShard allows or denies

- **effective_condition** (*Optional[policyglass.condition.EffectiveCondition]*) – The EffectiveCondition that needs to be met for this PolicyShard to apply

Return type None

dict(**args*, ***kwargs*)

Convert instance to dict representation of it.

Parameters

- ***args** – Arguments to Pydantic dict method.
- ****kwargs** – Arguments to Pydantic dict method.

Return type Dict[str, Any]

Overridden from BaseModel so that when converting conditions to dict they don't suffer from being unhashable when placed in a set.

difference(*other*, *dedupe_result=True*)

Calculate the difference between this and another object of the same type.

Effectively subtracts the inclusions of `other` from `self`. This is useful when applying denies (`other`) to allows (`self`).

Parameters

- **other** (*object*) – The object to subtract from this one.
- **dedupe_result** (*bool*) – Whether to deduplicate the resulting PolicyShards or not. Setting this to `False` will lead to many duplicates.

Raises **ValueError** – If `other` is not the same type as this object.

Return type List[*policyglass.policy_shard.PolicyShard*]

effect: str

effective_action: policyglass.effective_arp.EffectiveARP[*policyglass.action.Action*]

effective_condition: *policyglass.condition.EffectiveCondition*

effective_principal: policyglass.effective_arp.EffectiveARP[*policyglass.principal.Principal*]

effective_resource: policyglass.effective_arp.EffectiveARP[*policyglass.resource.Resource*]

property explain: str

Return a plain English representation of the policy shard.

Example

Simple PolicyShard explain.

```
>>> from policyglass import Policy
>>> policy = Policy(**{"Statement": [{"Effect": "Allow", "Action": "s3:*"}]})
>>> print([shard.explain for shard in policy.policy_shards])
['Allow action s3:* on resource * with principal AWS *.']
```

intersection(*other*)

Calculate the intersection between this object and another object of the same type.

Parameters **other** (*object*) – The object to intersect with this one.

Raises **ValueError** – if `other` is not the same type as this object.

Return type Optional[*policyglass.policy_shard.PolicyShard*]

issubset(*other*)

Whether this object contains all the elements of another object (i.e. is a subset of the other object).

Conditions: If both PolicyShards have conditions but are otherwise identical, self will be a subset of other if the other's conditions are are a subset of self's as this means that self is more restrictive and therefore carves out a subset of possiblilites in comparison with other.

Parameters **other** (*object*) – The object to determine if our object contains.

Raises **ValueError** – If the other object is not of the same type as this object.

Return type bool

union(*other*)

Combine this object with another object of the same type.

Parameters **other** (*object*) – The object to combine with this one.

Raises **ValueError** – If `other` is not the same type as this object.

Return type List[*policyglass.policy_shard.PolicyShard*]

dedupe_policy_shard_subsets(*shards*, *check_reverse=True*)

Dedupe policy shards that are subsets of each other.

Parameters

- **shards** (*Iterable[policyglass.policy_shard.PolicyShard]*) – The shards to deduplicate.
- **check_reverse** (*bool*) – Whether you want to check these shards in reverse as well (only disabled when alling itself).

Return type List[*policyglass.policy_shard.PolicyShard*]

dedupe_policy_shards(*shards*, *check_reverse=True*)

Dedupe policy shards that are subsets of each other and remove intersections.

Parameters

- **shards** (*Iterable[policyglass.policy_shard.PolicyShard]*) – The shards to deduplicate.
- **check_reverse** (*bool*) – Whether you want to check these shards in reverse as well (only disabled when calling itself).

Return type List[*policyglass.policy_shard.PolicyShard*]

explain_policy_shards(*shards*, *language='en'*)

Return a list of string explanations for a given list of PolicyShards.

Example

How to get the effective permissions of a policy as a plain English explanation.

```
>>> from policyglass import Policy, policy_shards_effect, explain_policy_shards
>>> policy = Policy(
...     **{
...         "Version": "2012-10-17",
...         "Statement": [
...             {
...                 "Effect": "Allow",
...                 "Action": ["s3:*"],
...                 "Resource": "*",
...             },
...             {
...                 "Effect": "Deny",
...                 "Action": ["s3:Get*"],
...                 "Resource": "*",
...             },
...         ],
...     }
... )
>>> explain_policy_shards(policy_shards_effect(policy.policy_shards))
['Allow action s3:* (except for s3:Get*) on resource * with principal AWS *.']
```

Parameters

- **shards** (*List[policyglass.policy_shard.PolicyShard]*) – The PolicyShards to explain.
- **language** (*str*) – The language of the explanation

Raises NotImplementedError – When an unsupported language is requested.

Return type List[str]

policy_shards_effect(*shards*)

Calculate the effect of merging allow and deny shards together.

Example

How to get the effective permissions of a policy as *PolicyShard* objects.

```
>>> from policyglass import Policy, policy_shards_effect, explain_policy_shards
>>> policy = Policy(
...     **{
...         "Version": "2012-10-17",
...         "Statement": [
...             {
...                 "Effect": "Allow",
...                 "Action": ["s3:*"],
...                 "Resource": "*",
...             },
...             {
```

(continues on next page)

(continued from previous page)

```
...                 "Effect": "Deny",
...                 "Action": ["s3:Get*"],
...                 "Resource": "*",
...             },
...         ],
...     }
... )
>>> policy_shards = policy.policy_shards
>>> policy_shards_effect(policy_shards)
[PolicyShard(effect='Allow',
    effective_action=EffectiveAction(inclusion=Action('s3:*'),
        exclusions=frozenset({Action('s3:Get*')})),
    effective_resource=EffectiveResource(inclusion=Resource('*'),
        exclusions=frozenset()),
    effective_principal=EffectivePrincipal(inclusion=Principal(type='AWS', value='*
↪'),
        exclusions=frozenset()),
    effective_condition=EffectiveCondition(inclusions=frozenset(),
        exclusions=frozenset()))]
```

Parameters **shards** (*List[policyglass.policy_shard.PolicyShard]*) – The shards to caclulate the effect of.

Return type List[*policyglass.policy_shard.PolicyShard*]

policy_shards_to_json(*shards*, *exclude_defaults=False*, ***kwargs*)

Convert a list of *PolicyShard* objects to JSON.

Example

How to get the effective permissions of a policy as json.

```
>>> from policyglass import Policy, policy_shards_effect, policy_shards_to_json
>>> policy = Policy(
...     **{
...         "Version": "2012-10-17",
...         "Statement": [
...             {
...                 "Effect": "Allow",
...                 "Action": ["s3:*"],
...                 "Resource": "*",
...             },
...             {
...                 "Effect": "Deny",
...                 "Action": ["s3:Get*"],
...                 "Resource": "*",
...             },
...         ],
...     }
... )
>>> policy_shards = policy.policy_shards
```

(continues on next page)

(continued from previous page)

```
>>> output = policy_shards_to_json(
...     policy_shards_effect(policy_shards),
...     indent=2,
...     exclude_defaults=True
... )
>>> print(output)
[
    {
        "effective_action": {
            "inclusion": "s3:*",
            "exclusions": [
                "s3:Get*"
            ]
        },
        "effective_resource": {
            "inclusion": "*"
        },
        "effective_principal": {
            "inclusion": {
                "type": "AWS",
                "value": "*"
            }
        }
    }
]
```

Parameters

- **shards** (*List[policyglass.policy_shard.PolicyShard]*) – The list of shards to convert.
- **exclude_defaults** – Whether to exclude default values (e.g. empty lists) from the output.
- ****kwargs** – keyword arguments passed on to `json.dumps()`

Return type str

5.3 Statement

Statement class.

class Effect

Allow or Deny.

class Statement(*, *Effect*, *Action=None*, *NotAction=None*, *Resource=None*, *NotResource=None*, *Principal=None*, *NotPrincipal=None*, *Condition=None*)

A Policy Statement.

Parameters

- **Effect** (policyglass.statement.Effect) –
- **Action** (*List[policyglass.action.Action]*) –
- **NotAction** (*List[policyglass.action.Action]*) –

- **Resource** (*List[policyglass.resource.Resource]*) –
- **NotResource** (*List[policyglass.resource.Resource]*) –
- **Principal** (policyglass.principal.PrincipalCollection) –
- **NotPrincipal** (policyglass.principal.PrincipalCollection) –
- **Condition** (policyglass.condition.RawConditionCollection) –

Return type None

class Config

Configure the Pydantic BaseModel.

alias_generator()

Convert a snake_case string into a PascalCase string.

Parameters **string** (*str*) – The string to convert to PascalCase.

Return type str

action: Optional[List[*policyglass.action.Action*]]

condition: Optional[*policyglass.condition.RawConditionCollection*]

effect: *policyglass.statement.Effect*

classmethod ensure_action_list(*v*)

Parameters **v** (*policyglass.statement.T*) –

Return type List[*policyglass.action.Action*]

classmethod ensure_condition_value_list(*v*)

Parameters **v** (*Dict[policyglass.condition.ConditionKey, Dict[policyglass.condition.ConditionOperator, Union[policyglass.condition.ConditionValue, List[policyglass.condition.ConditionValue]]]]*) –

Return type *policyglass.condition.RawConditionCollection*

classmethod ensure_principal_dict(*v*)

Parameters **v** (*Union[policyglass.principal.PrincipalValue, Dict[policyglass.principal.PrincipalType, Union[policyglass.principal.PrincipalValue, List[policyglass.principal.PrincipalValue]]]]*) –

Return type *policyglass.principal.PrincipalCollection*

classmethod ensure_resource_list(*v*)

Parameters **v** (*policyglass.statement.T*) –

Return type List[*policyglass.resource.Resource*]

not_action: Optional[List[*policyglass.action.Action*]]

not_principal: Optional[*policyglass.principal.PrincipalCollection*]

not_resource: Optional[List[*policyglass.resource.Resource*]]

policy_json()

Return type str

property policy_shards: List[*policyglass.policy_shard.PolicyShard*]

principal: Optional[*policyglass.principal.PrincipalCollection*]

resource: Optional[List[*policyglass.resource.Resource*]]

5.4 Action

Action class.

class Action

Actions are case insensitive.

> "The prefix and the action name are case insensitive"
>
> —IAM JSON policy elements: Action

issubset(*other*)

Whether this object contains all the elements of another object (i.e. is a subset of the other object).

Parameters **other** (*object*) – The object to determine if our object contains.

Raises **ValueError** – If the other object is not of the same type as this object.

Return type bool

class EffectiveAction(*inclusion*, *exclusions=None*)

EffectiveActions are the representation of the difference between an Action and its exclusion.

The allowed actions is the difference (subtraction) of the excluded Actions from the included action.

exclusions: FrozenSet[policyglass.effective_arp.T]

Exclusions must always be a subset of the include and must not be subsets of each other

inclusion: policyglass.effective_arp.T

Inclusion must be a superset of any exclusions

5.5 Resource

Resource class.

class EffectiveResource(*inclusion*, *exclusions=None*)

EffectiveResources are the representation of the difference between an Resource and its exclusion.

The allowed Resource is the difference (subtraction) of the excluded Resources from the included Resource.

exclusions: FrozenSet[policyglass.effective_arp.T]

Exclusions must always be a subset of the include and must not be subsets of each other

inclusion: policyglass.effective_arp.T

Inclusion must be a superset of any exclusions

class Resource

A resource ARN may be case sensitive or case insensitive depending on the resource type.

property arn_elements: List[str]

Return a list of arn elements, replacing blanks with *.

issubset(*other*)
Whether this object contains all the elements of another object (i.e. is a subset of the other object).

Parameters **other** (*object*) – The object to determine if our object contains.

Raises ValueError – If the other object is not of the same type as this object.

Return type bool

5.6 Principal

Principal classes.

class EffectivePrincipal(*inclusion*, *exclusions=None*)
EffectivePrincipals are the representation of the difference between an Principal and its exclusion.

The allowed Principal is the difference (subtraction) of the excluded Principals from the included Principal.

exclusions: FrozenSet[policyglass.effective_arp.T]
Exclusions must always be a subset of the include and must not be subsets of each other

inclusion: policyglass.effective_arp.T
Inclusion must be a superset of any exclusions

class Principal(*type*, *value*)
A class which represents a single Principal including its type.

Objects of this type are typically generated by the *Statement* class.

Parameters

- **type** (policyglass.principal.PrincipalType) –
- **value** (policyglass.principal.PrincipalValue) –

Return type None

__init__(*type*, *value*)
Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

Parameters

- **type** (policyglass.principal.PrincipalType) –
- **value** (policyglass.principal.PrincipalValue) –

Return type None

property account_id: Optional[str]
Return the account id of this Principal if there is one.

property arn_elements: List[str]
Return a list of arn elements, replacing blanks with "".

property is_account: bool
Return true if the prinncipal is an account.

issubset(*other*)
Whether this object contains all the elements of another object (i.e. is a subset of the other object).

Parameters **other** (*object*) – The object to determine if our object contains.

Raises ValueError – If the other object is not of the same type as this object.

Return type bool

type: *policyglass.principal.PrincipalType*
Principal Type

value: *policyglass.principal.PrincipalValue*
Principal value

class PrincipalCollection
A collection of Principals of different types, unique to PolicyGlass.

property principals: List[*policyglass.principal.Principal*]

class PrincipalType
A principal type, e.g. Federated or AWS.

See AWS JSON policy elements: Principal for more.

class PrincipalValue
An ARN, wildcard, or other appropriate value of a policy Principal.

See AWS JSON policy elements: Principal for more.

5.7 Condition

Statement Condition classes.

class Condition(*key*, *operator*, *values*)
A representation of part of a statement condition in order to facilitate comparison.

Parameters

- **key** (policyglass.condition.ConditionKey) –
- **operator** (policyglass.condition.ConditionOperator) –
- **values** (*List[*policyglass.condition.ConditionValue*]*) –

Return type None

__init__(*key*, *operator*, *values*)
Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

Parameters

- **key** (policyglass.condition.ConditionKey) –
- **operator** (policyglass.condition.ConditionOperator) –
- **values** (*List[*policyglass.condition.ConditionValue*]*) –

Return type None

classmethod factory(*condition_collection*)

Parameters **condition_collection** (policyglass.condition.RawConditionCollection) –

Return type FrozenSet[*policyglass.condition.Condition*]

key: *policyglass.condition.ConditionKey*

operator: *policyglass.condition.ConditionOperator*

property reverse: *policyglass.condition.Condition*

Return a new condition which is the opposite of this condition.

Raises ValueError – If the operator is a type that cannot be reversed.

values: List[*policyglass.condition.ConditionValue*]

class ConditionKey

Condition Keys are case insensitive.

"Condition key names are not case-sensitive." - IAM Reference Policy Elements

class ConditionOperator

Condition Operator.

See IAM JSON policy elements: Condition operators for more.

class ConditionValue

Condition values may or may not be case sensitive depending on the operator.

class EffectiveCondition(*inclusions=None*, *exclusions=None*)

A pair of sets for inclusions and exclusion conditions.

Parameters

- **inclusions** (*FrozenSet[policyglass.condition.Condition]*) –
- **exclusions** (*FrozenSet[policyglass.condition.Condition]*) –

Return type None

__init__(*inclusions=None*, *exclusions=None*)

Convert `exclusions` to `inclusions` if possible.

The only type of Condition that really exists in AWS policies is the `inclusions`. The `exclusions` are created only when conditions on a `Deny` statement have operators that cannot be reversed. The reversal is required in order to fold a Deny condition into an Allow condition.

Parameters

- **inclusions** (*Optional[FrozenSet[policyglass.condition.Condition]]*) – The conditions that must be met.
- **exclusions** (*Optional[FrozenSet[policyglass.condition.Condition]]*) – The conditions that must NOT be met.

Return type None

dict(**args*, ***kwargs*)

Convert instance to dict representation of it.

Parameters

- ***args** – Arguments to Pydantic dict method.
- ****kwargs** – Arguments to Pydantic dict method.

Return type Dict[str, Any]

Overridden from BaseModel so that when converting conditions to dict they don't suffer from being unhashable when placed in a set.

exclusions: FrozenSet[*policyglass.condition.Condition*]

Conditions which must NOT be met

inclusions: FrozenSet[*policyglass.condition.Condition*]
Conditions which must be met

intersection(*other*)
Calculate the intersection between this object and another object of the same type.

Parameters **other** (*object*) – The object to intersect with this one.

Raises **ValueError** – if `other` is not the same type as this object.

Return type *policyglass.condition.EffectiveCondition*

property reverse: *policyglass.condition.EffectiveCondition*
Reverse the effect of this EffectiveCondition.

union(*other*)
Combine this object with another object of the same type.

Parameters **other** (*object*) – The object to combine with this one.

Raises **ValueError** – If `other` is not the same type as this object.

Return type *policyglass.condition.EffectiveCondition*

```
OPERATOR_REVERSAL_INDEX = {ConditionOperator('ArnEquals'):
ConditionOperator('ArnNotEquals'), ConditionOperator('ArnEqualsIfExists'):
ConditionOperator('ArnNotEqualsIfExists'), ConditionOperator('ArnLike'):
ConditionOperator('ArnNotLike'), ConditionOperator('ArnLikeIfExists'):
ConditionOperator('ArnNotLikeIfExists'), ConditionOperator('ArnNotEquals'):
ConditionOperator('ArnEquals'), ConditionOperator('ArnNotEqualsIfExists'):
ConditionOperator('ArnEqualsIfExists'), ConditionOperator('ArnNotLike'):
ConditionOperator('ArnLike'), ConditionOperator('ArnNotLikeIfExists'):
ConditionOperator('ArnLikeIfExists'), ConditionOperator('DateEquals'):
ConditionOperator('DateNotEquals'), ConditionOperator('DateEqualsIfExists'):
ConditionOperator('DateNotEqualsIfExists'), ConditionOperator('DateGreaterThan'):
ConditionOperator('DateLessThanEquals'), ConditionOperator('DateGreaterThanEquals'):
ConditionOperator('DateLessThan'), ConditionOperator('DateGreaterThanEqualsIfExists'):
ConditionOperator('DateLessThanIfExists'), ConditionOperator('DateGreaterThanIfExists'):
ConditionOperator('DateLessThanEqualsIfExists'), ConditionOperator('DateLessThan'):
ConditionOperator('DateGreaterThanEquals'), ConditionOperator('DateLessThanEquals'):
ConditionOperator('DateGreaterThan'), ConditionOperator('DateLessThanEqualsIfExists'):
ConditionOperator('DateGreaterThanIfExists'), ConditionOperator('DateLessThanIfExists'):
ConditionOperator('DateGreaterThanEqualsIfExists'), ConditionOperator('DateNotEquals'):
ConditionOperator('DateEquals'), ConditionOperator('DateNotEqualsIfExists'):
ConditionOperator('DateEqualsIfExists'), ConditionOperator('IpAddress'):
ConditionOperator('NotIpAddress'), ConditionOperator('IpAddressIfExists'):
ConditionOperator('NotIpAddressIfExists'), ConditionOperator('NotIpAddress'):
ConditionOperator('IpAddress'), ConditionOperator('NotIpAddressIfExists'):
ConditionOperator('IpAddressIfExists'), ConditionOperator('NumericEquals'):
ConditionOperator('NumericNotEquals'), ConditionOperator('NumericEqualsIfExists'):
ConditionOperator('NumericNotEqualsIfExists'), ConditionOperator('NumericGreaterThan'):
ConditionOperator('NumericLessThanEquals'),
ConditionOperator('NumericGreaterThanEquals'):  ConditionOperator('NumericLessThan'),
ConditionOperator('NumericGreaterThanEqualsIfExists'):
ConditionOperator('NumericLessThanIfExists'),
ConditionOperator('NumericGreaterThanIfExists'):
ConditionOperator('NumericLessThanEqualsIfExists'), ConditionOperator('NumericLessThan'):
ConditionOperator('NumericGreaterThanEquals'),
ConditionOperator('NumericLessThanEquals'):  ConditionOperator('NumericGreaterThan'),
ConditionOperator('NumericLessThanEqualsIfExists'):
ConditionOperator('NumericGreaterThanIfExists'),
ConditionOperator('NumericLessThanIfExists'):
ConditionOperator('NumericGreaterThanEqualsIfExists'),
ConditionOperator('NumericNotEquals'):  ConditionOperator('NumericEquals'),
ConditionOperator('NumericNotEqualsIfExists'):
ConditionOperator('NumericEqualsIfExists'), ConditionOperator('StringEquals'):
ConditionOperator('StringNotEquals'), ConditionOperator('StringEqualsIfExists'):
ConditionOperator('StringNotEqualsIfExists'),
ConditionOperator('StringEqualsIgnoreCase'):
ConditionOperator('StringNotEqualsIgnoreCase'),
ConditionOperator('StringEqualsIgnoreCaseIfExists'):
ConditionOperator('StringNotEqualsIgnoreCaseIfExists'), ConditionOperator('StringLike'):
ConditionOperator('StringNotLike'), ConditionOperator('StringLikeIfExists'):
ConditionOperator('StringNotLikeIfExists'), ConditionOperator('StringNotEquals'):
ConditionOperator('StringEquals'), ConditionOperator('StringNotEqualsIfExists'):
ConditionOperator('StringEqualsIfExists'),
ConditionOperator('StringNotEqualsIgnoreCase'):
ConditionOperator('StringEqualsIgnoreCase'),
ConditionOperator('StringNotEqualsIgnoreCaseIfExists'):
ConditionOperator('StringEqualsIgnoreCaseIfExists'), ConditionOperator('StringNotLike'):
ConditionOperator('StringLike'), ConditionOperator('StringNotLikeIfExists'),
ConditionOperator('StringLikeIfExists')}
```

A list of operators and their opposite.

class RawConditionCollection

A representation of a statement condition.

property conditions: FrozenSet[*policyglass.condition.Condition*]

Return a list of Condition Shards.

5.8 Understanding Effective Conditions

Policy conditions, when they exist, are always restrictions on the scenarios in which a policy applies. Every *PolicyShard* object will have a *EffectiveCondition* object, even if the *EffectiveCondition* has no `inclusions` or `exclusions` specified.

5.8.1 What is an inclusion/exclusion?

An *EffectiveCondition* `inclusion` is a *Condition* which must be true, for a *PolicyShard* to apply. An *EffectiveCondition* `exclusion` is a *Condition* which must be **false**, for a *PolicyShard* to apply.

```
>>> from policyglass import PolicyShard, EffectiveAction, Action, EffectiveResource,␣
↪Resource, EffectivePrincipal, Principal, EffectiveCondition, Condition
>>> effective_condition = EffectiveCondition(
...     inclusions=frozenset({
...         Condition("aws:PrincipalOrgId", "StringEquals", ["o-123456"]),
...     }),
...     exclusions=frozenset({
...         Condition(key="TestKey", operator="BinaryEquals", values=[
↪"QmluYXJ5VmFsdWVJbkJhc2U2NA=="])
...    }),
... )
>>> policy_shard = PolicyShard(
...     effect="Allow",
...     effective_action=EffectiveAction(Action("*")),
...     effective_resource=EffectiveResource(Resource("*")),
...     effective_principal=EffectivePrincipal(Principal("AWS", "*")),
...     effective_condition=effective_condition
... )
```

This `effective_condition`'s `inclusions` dictate that for `Action`, `Resource` and `Principal` to be allowed, then at the time the API call takes place the following be true:

1. `aws:PrincipalOrgId` must `StringEquals` a value of `o-123456`.
2. `TestKey` must **NOT** `BinaryEquals` a value of `QmluYXJ5VmFsdWVJbkJhc2U2NA==`

5.8.2 When would an exclusion occur?

An *EffectiveCondition* `exclusion` is quite a rare phenomenon. Normally when `Deny` *PolicyShard* conditions are folded into `Allow` *PolicyShard* objects, they are reversed using the *reverse* attribute.

For example `StringNotEquals` on a `Deny` PolicyShard will become `StringEquals` on an `Allow` PolicyShard. This simplifies the intelligibility of the `Allow` shards significantly.

When a `Deny` statement has a condition that cannot be reversed (e.g. `BinaryEquals` for which there is no corresponding `BinaryNotEquals`) then the condition must be placed into the `exclusions` of the *effective_condition* of the `Allow` PolicyShard.

5.9 Understanding Effective Actions

In PolicyGlass we express ARPs (*Action Resource policyglass.principal.Principal*) as though they are potentially infinite sets.

In reality they are finite sets because there are only a finite number of allowed actions, resources, or principals. However because actions are being constantly updated by AWS, and new resources and princiapls are being created all the time, we here treat them as infinite sets because their extent is unknowable by us when we are parsing the policy.

5.9.1 Components of an EffectiveAction

An *EffectiveAction* object has two components:

1. *inclusion*
2. *exclusions*

The inclusions indicate the *Action* that this effective action applies to and the exclusions indicate the actions that this effective action *does not* apply to.

At its simplest an effective action is just an inclusion, which you can think of as a Venn diagram containing `S3:*`.

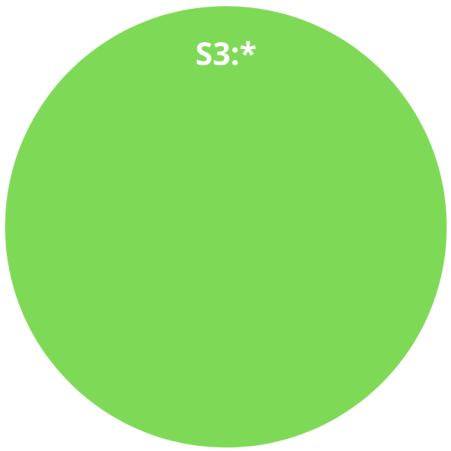

Fig. 1: EffectiveAction without exclusion

Then if you have an exclusion of `S3:Get*` you can think of this as a hole punched in the Venn diagram.

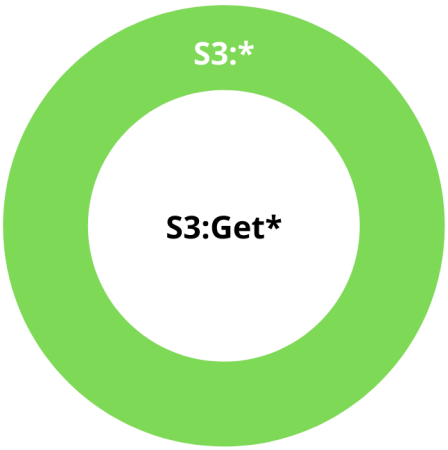

Fig. 2: EffectiveAction with exclusion

The area in the middle indicating that `S3:Get*` is not included in the effective action.

5.9.2 Difference

The *difference* between set *x* and set *y* is the elements that are contained in set *x* that are not contained in set *y*. In essence it's a subtraction. Remove the elements in set *y* from set *x* and you have the difference.

Simple

Let's say we calculate the difference between two effective actions like so.

```
>>> from policyglass import EffectiveAction, Action
>>> x = EffectiveAction(inclusion=Action("S3:*"))
>>> y = EffectiveAction(inclusion=Action("S3:Get*"))
>>> x.difference(y)
[EffectiveAction(inclusion=Action('S3:*'), exclusions=frozenset({Action('S3:Get*')}))]
```

The result is that the inclusion from *y* is added to the *exclusions* of *x*.

- `S3:*` is the `inclusion` from `x`
- `S3:Get*` is the `inclusion` from `y`

The inclusion from *x* is added as an exclusion of *y* is because our Actions are essentially infinite sets. The wildcard at the end of `S3:*` could extend to an infinitely long string for all we know, so we can't create an *Action* that expresses *S3:** but not *S3:Get** so we must add it as an exclusion in an *EffectiveAction*.

This is the reason *EffectiveAction* s exist, so we can express the intersection of the complement of ininite set B with inifite set A.

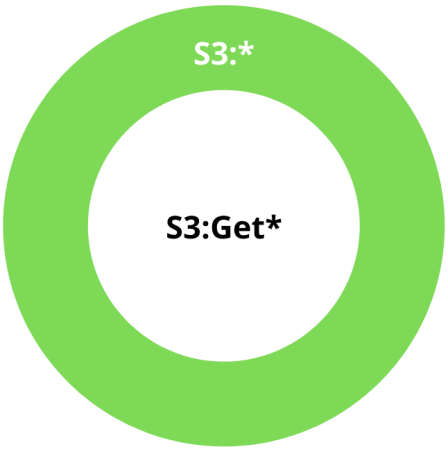

Fig. 3: Simple Difference

Complex

Let's say we have two effective actions we want to diff. One is just `S3:*` and the other is `S3:Get*` except for `S3:GetObject`. To diff these we want to subtract `S3:Get*` from `S3:*` but leave `S3:GetObject` in place.

```
>>> from policyglass import EffectiveAction, Action
>>> x = EffectiveAction(inclusion=Action("S3:*"))
>>> y = EffectiveAction(inclusion=Action("S3:Get*"), exclusions=frozenset({Action(
↪"S3:GetObject")}))
>>> print(x.difference(y))
[EffectiveAction(inclusion=Action('S3:*'), exclusions=frozenset({Action('S3:Get*')})),
 EffectiveAction(inclusion=Action('S3:GetObject'), exclusions=frozenset())]
```

Let's unpack what happened here.

1. We added the *inclusion* (`S3:get*`) from *y* to the exclusions of *x*
2. We returned a new effective action that is just `S3:GetObject`

- `S3:*` is our `inclusion` from `x`
- `S3:Get*` is our `inclusion` from `y`
- `S3:GetObject` is our `exclusion` from `y`

In the above Venn diagram we're showing that the difference between the two effective actions is to include `S3:*` except `S3:Get*` but still include `S3:GetObject`. We can't have an inclusion inside an exclusion so we represent this by adding another effective action object to represent the inclusion.

Outputting two effective actions makes a list of *PolicyShard* objects much easier to understand because you will end up with two shards (one for each effective action) rather than one super hard to understand shard that has an action inclusion inside an action exclusion inside an action inclusion.

Remember that the exclusions in an EffectiveAction are negations, they are holes punched in what's allowed. As a result, what is in the exclusion of *y* should **not** be removed from *x* because it's explicitly not part of *y*.

Because we can't express the fact that we want to exclude B and C but **include** A in our result, we have to return two separate *EffectiveAction* s, one which includes A but the entirety of B, and another that just includes D.

Fig. 4: Complex Difference (theoretical)

Fig. 5: Complex Difference (actual output)

5.10 Understanding Policy Shards

5.10.1 Dedupe & Merge

PolicyShard objects need to go through two phases to correct their sizes.

1. Dedupe using *dedupe_policy_shard_subsets()*
2. Merge using *dedupe_policy_shards()*

The first collapses all PolicyShards which are subsets of each other into each other, in other words eliminating all smaller PolicyShards that can fit into bigger PolicyShards.

The second reverses that where necessary, identifying shards that are not subsets of each other but nonetheless have some intersection and therefore duplicate the permissions space.

When does a PolicyShard intesect without being a subset?

This is a departure from EffectiveARPs (Action, Resource, Principal) objects which by contrast cannot intersect without being subsets.

Let's consider this scenario

```
>>> from policyglass import PolicyShard
>>> from policyglass.action import Action, EffectiveAction
>>> from policyglass.condition import Condition, EffectiveCondition
>>> from policyglass.principal import EffectivePrincipal, Principal
>>> from policyglass.resource import EffectiveResource, Resource
>>> shard_a = PolicyShard(
...     effect="Allow",
...     effective_action=EffectiveAction(inclusion=Action("s3:*"), exclusions=frozenset(
↪{Action("s3:PutObject")})),
...     effective_resource=EffectiveResource(inclusion=Resource("*")),
...     effective_principal=EffectivePrincipal(inclusion=Principal(type="AWS", value="*
↪")),
...     effective_condition=EffectiveCondition(frozenset(
...         {Condition(key="aws:PrincipalOrgId", operator="StringNotEquals", values=["o-
↪123456"])}
...     )),
... )
>>> shard_b = PolicyShard(
...     effect="Allow",
...     effective_action=EffectiveAction(inclusion=Action("s3:*")),
...     effective_resource=EffectiveResource(inclusion=Resource("*")),
...     effective_principal=EffectivePrincipal(inclusion=Principal(type="AWS", value="*
↪")),
...     effective_condition=EffectiveCondition(frozenset(
...         {
...             Condition(key="aws:PrincipalOrgId", operator="StringNotEquals", values=[
↪"o-123456"]),
...             Condition(key="s3:x-amz-server-side-encryption", operator="StringEquals",
↪ values=["AES256"]),
...         }
...     )),
... )
```

Shard A

1. Has a single condition
2. Excludes `s3:PutObject`

Shard B

1. Has two conditions, one of which is the same as Shard A
2. Does not exclude `s3:PutObject`

This means that.

1. Because Shard A and Shard B both have conditions they can never be considered subsets of one another even during the decomposition process
2. They do intersect because every part of `s3:*` apart from `s3:PutObject` is less restrictively allowed by Shard A
3. We want to reduce the scope of Shard B to just `s3:PutObject`

To do this we use *dedupe_policy_shards()*

```
>>> from policyglass.policy_shard import dedupe_policy_shards
>>> shard_b_delineated, shard_a_delineated = dedupe_policy_shards([shard_a, shard_b])
>>> assert shard_a_delineated == PolicyShard(
...     effect='Allow',
...     effective_action=EffectiveAction(inclusion=Action('s3:*'), exclusions=frozenset(
↪{Action('s3:PutObject')})),
...     effective_resource=EffectiveResource(inclusion=Resource('*')),
...     effective_principal=EffectivePrincipal(inclusion=Principal(type='AWS', value='*
↪')),
...     effective_condition=EffectiveCondition(frozenset(
...         {Condition(key='aws:PrincipalOrgId', operator='StringNotEquals', values=['o-
↪123456'])}
...     )),
... )
>>> assert shard_b_delineated == PolicyShard(
...     effect='Allow',
...     effective_action=EffectiveAction(inclusion=Action('s3:PutObject')),
...     effective_resource=EffectiveResource(inclusion=Resource('*')),
...     effective_principal=EffectivePrincipal(inclusion=Principal(type='AWS', value='*
↪')),
...     effective_condition=EffectiveCondition(frozenset({
...         Condition(key='aws:PrincipalOrgId', operator='StringNotEquals', values=['o-
↪123456']),
...         Condition(key='s3:x-amz-server-side-encryption', operator='StringEquals',␣
↪values=['AES256'])
...     })),
... )
```

You'll notice that the intersection has been removed, as Shard B now only has `s3:PutObject` as the rest of `s3:*` was covered by Shard A.

CHAPTER

SIX

POLICYGLASS

Documentation: policyglass.cloudwanderer.io
GitHub: https://github.com/CloudWanderer-io/PolicyGlass

PolicyGlass allows you to analyse one or more AWS policies' effective permissions in aggregate, by restating them in the form of PolicyShards which are always Allow, never Deny.

PolicyGlass will **always** result in only allow `PolicyShard` objects, no matter how complex the policy. This makes understanding the effect of your policies programmatically a breeze.

6.1 Try it out

Try out custom policies quickly without installing anything with the PolicyGlass Sandbox.

6.2 Installation

```
pip install policyglass
```

6.3 Usage

Let's take two policies, *a* and *b* and pit them against each other.

```
>>> from policyglass import Policy, policy_shards_effect
>>> policy_a = Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Allow",
...             "Action": [
...                 "s3:*"
...             ],
...             "Resource": "*"
...         }
...     ]
... })
>>> policy_b = Policy(**{
...     "Version": "2012-10-17",
...     "Statement": [
...         {
...             "Effect": "Deny",
...             "Action": [
...                 "s3:*"
...             ],
...             "Resource": "arn:aws:s3:::examplebucket/*"
...         }
...     ]
... })
>>> policy_shards = [*policy_a.policy_shards, *policy_b.policy_shards]
>>> effect = policy_shards_effect(policy_shards)
>>> effect
[PolicyShard(effect='Allow',
   effective_action=EffectiveAction(inclusion=Action('s3:*'),
      exclusions=frozenset()),
   effective_resource=EffectiveResource(inclusion=Resource('*'),
      exclusions=frozenset({Resource('arn:aws:s3:::examplebucket/*')})),
   effective_principal=EffectivePrincipal(inclusion=Principal(type='AWS', value='*'),
      exclusions=frozenset()),
   effective_condition=EffectiveCondition(inclusions=frozenset(),␣
↪exclusions=frozenset()))]
```

Two policies, two statements, resulting in a single allow `PolicyShard`. More complex policies will result in multiple shards, but they will always be **allows**, no matter how complex the policy.

You can also make them human readable!

```
>>> from policyglass import explain_policy_shards
>>> explain_policy_shards(effect)
['Allow action s3:* on resource * (except for arn:aws:s3:::examplebucket/*) with␣
↪principal AWS *.']
```

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

D

E

F

I

J

K

M

N

O

P

R

S

T

U

V