Response Functions in AWS

All information on this page is automatically updated with every build, typically every ~24 hours.

Remediation and Response in AWS comes with the following built-in response functions:

Show All ↓

AWS_EBS_001 - Generate EBS snapshot for recovery purposes

Identifies Elastic Block Store (EBS) volumes that were not backed up via snapshots in the last 15 days, a critical period for maintaining data security and system stability. If the function finds an EBS volume lacking a snapshot within this timeframe, it automatically initiates the creation of a new snapshot.
Mapped Cloud Configuration Rule: EBS-005

AWS_EBS_001

```
"""
Required Permissions:
- ec2:CreateSnapshot
- ec2:DescribeSnapshots
- ec2:CreateTags
"""

import boto3
from botocore.exceptions import ClientError
from datetime import date

from .. import auto_tagging
from .. import utils
from .. import constants

# Options:
#
# Snapshot age in days
#
snapshot_age = 15
```

```
def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    volume_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    ec2 = session.client("ec2", region_name=region)

    try:
        snapshot = ec2.describe_snapshots(
            Filters=[{"Name": "volume-id", "Values": [volume_id]}]
        )["Snapshots"]
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return
```

AWS_EBS_002 - Remove public attribute on an EBS snapshot

To avoid exposing personal and sensitive data, we recommend against sharing your EBS snapshots with all AWS accounts. This function removes the public attribute on an EBS snapshot
Mapped Cloud Configuration Rule: EBS-007

AWS_EBS_002

```
"""
Required Permissions:
- ec2:DescribeSnapshotAttribute
- ec2:ModifySnapshotAttribute
- ec2:CreateTags
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging
from .. import constants
from .. import utils
```

```
def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    snapshot_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    ec2 = session.client("ec2", region_name=region)

    try:
        snap_attrib = ec2.describe_snapshot_attribute(
            Attribute="createVolumePermission", SnapshotId=snapshot_id
        )
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    vol_perms = (
        snap_attrib["CreateVolumePermissions"]
        if ("CreateVolumePermissions" in snap_attrib)
        else ""
    )
```

AWS_EBS_003 - Migrate EBS volume from GP2 to GP3

Migrates an EBS volume from GP2 to GP3 inline
Mapped Cloud Configuration Rule: [EBS-009]

AWS_EBS_003

```
"""
Required Permissions:
- ec2:ModifyVolume
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging
from .. import utils
from .. import constants
```

```
# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, alert, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    volume_id = alert['external_id']
    region = alert['region']
    scan_id = alert['scanId']
    presigned_url = alert['presignURL']
    subscription_id = alert['subscription']['id']

    ec2 = session.client('ec2', region_name=region)

    try:
        describe_volumes_response = ec2.describe_volumes(VolumeIds=
[volume_id])
    except ClientError as e:
        response_action_message = e.response['Error']['Message']
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(presigned_url, scan_id,
response_action_status, response_action_message)
        return

    current_volume_type = describe_volumes_response.get('Volumes',
[{'VolumeType':'Unknown'}])[0]['VolumeType']
    print(f"Current VolumeType for {volume_id} is:
{current_volume_type}")

    try:
        ec2.modify_volume(
```

AWS_EBS_004 - Delete unattached EBS volume created more than 7 days ago

Deletes unattached EBS volumes that are more than 7 days old
Mapped Cloud Configuration Rule: EBS-006

AWS_EBS_004

```
"""
Required Permissions:
- ec2:DescribeVolumes
- ec2:DeleteVolume
"""

import boto3
```

```
from botocore.exceptions import ClientError

from .. import utils
from .. import constants

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, alert, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    volume_id = alert['external_id']
    region = alert['region']
    scan_id = alert['scanId']
    presigned_url = alert['presignURL']

    ec2 = session.client('ec2', region_name=region)

    # get volume info and double check it has no attachments
    try:
        volume_info = ec2.describe_volumes(
            Filters=[
                {
                    'Name': 'volume-id',
                    'Values': [
                        volume_id,
                    ],
                }
            ]
        )

    except ClientError as e:
        response_action_message = e.response['Error']['Message']
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(presigned_url, scan_id,
response_action_status, response_action_message)
```

AWS_EC2_029 - Enforce IMDSv2 on EC2

Verifies and enforces the configuration of EC2 instance metadata to utilize Metadata Service Version 2 (IMDSv2), transitioning from optional or not required status to required
Mapped Cloud Configuration Rule: EC2-004

AWS_EC2_029

```
"""
Required Permissions:
```

```
- ec2:DescribeInstances
- ec2:ModifyInstanceMetadataOptions
- ec2:CreateTags
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    instance_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    ec2 = session.client("ec2", region_name=region)

    print(
        "Checking if IMDSv2 setting (MetadataOptions.HttpTokens) is not
set to 'required'"
    )
    try:
        instance = ec2.describe_instances(InstanceIds=[instance_id])
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    if len(instance) > 0:
        imdsv2_setting = instance["Reservations"][0]["Instances"][0]
["MetadataOptions"][
            "HttpTokens"
        ]

    else:
```

AWS_EC2_031 - Remove unrestricted access to certain ports from security group

Eliminates any unrestricted rule to prevent global access (any IP address) to TCP/UDP ports or ICMP within a security group

Mapped Cloud Configuration Rules: VPC-012, VPC-014, VPC-015, VPC-016, VPC-017, VPC-018, VPC-019, VPC-020, VPC-021, VPC-022, VPC-023, VPC-024, VPC-025, VPC-026, VPC-027, VPC-028, VPC-029, VPC-030, VPC-031, VPC-032, VPC-033, VPC-034, Firewall-001, Firewall-003, Firewall-004, Firewall-005, Firewall-006

AWS_EC2_031

```
"""
Required Permissions:
- ec2:DescribeSecurityGroups
- ec2:RevokeSecurityGroupIngress
- ec2:CreateTags
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils

GLOBAL_CIDR_IPV4 = "0.0.0.0/0"
GLOBAL_CIDR_IPV6 = "::/0"

ANY_PORT = -1
ANY_PROTOCOL = "-1"
PORT_HIGHER_THAN_1024 = (
    -1024
)  # using the minus range for extra features :) AWS is using it in
ANY_PORT so..

CCR_PORT_MAP = {
    "VPC-012": [{"protocol": ANY_PROTOCOL, "port": ANY_PORT}],
    "VPC-014": [{"protocol": "tcp", "port": 3389}],
    "VPC-015": [{"protocol": "tcp", "port": 22}],
    "VPC-016": [{"protocol": "tcp", "port": 23}],
    "VPC-017": [{"protocol": "tcp", "port": 5500}],
    "VPC-018": [{"protocol": "tcp", "port": 5800}, {"protocol": "tcp",
"port": 5900}],
    "VPC-019": [{"protocol": "tcp", "port": 135}],
    "VPC-020": [{"protocol": "tcp", "port": 445}],
    "VPC-021": [
        {"protocol": "tcp", "port": 139},
        {"protocol": "tcp", "port": 445},
        {"protocol": "udp", "port": 137},
        {"protocol": "udp", "port": 138},
        {"protocol": "udp", "port": 445},
    ],
    "VPC-022": [{"protocol": "tcp", "port": 53}],
    "VPC-023": [{"protocol": "udp", "port": 53}],
    "VPC-024": [{"protocol": "tcp", "port": 21}],
    "VPC-025": [{"protocol": "tcp", "port": 20}],
    "VPC-026": [{"protocol": "tcp", "port": 4333}, {"protocol": "udp",
"port": 4333}],
```

```
    "VPC-027": [{"protocol": "tcp", "port": 3306}],
```

AWS_EC2_RESPONSE_001 - Suspend/Shutdown compute instance

Stop an EC2 instance for an incident response.
Mapped target entity type for generic response function: `virtualMachine`

AWS_EC2_RESPONSE_001

```
"""
Required permissions:
ec2:DescribeInstances
ec2:StopInstances
"""

import boto3
from botocore.exceptions import ClientError, WaiterError

from .. import auto_tagging, constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    ec2_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    ec2_client = session.client("ec2", region_name=region)

    try:
        describe_response = ec2_client.describe_instances(InstanceIds=
[ec2_id])

    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return
```

```
    if (
        ("Reservations" not in describe_response)
        or (len(describe_response["Reservations"]) < 1)
        or ("Instances" not in describe_response["Reservations"][0])
    ):
```

AWS_EC2_RESPONSE_002 - Restart compute instance

Restart an EC2 instance for an incident response.
Mapped target entity type for generic response function: `virtualMachine`

AWS_EC2_RESPONSE_002

```
"""
Required permissions:
ec2:DescribeInstances
ec2:RebootInstances
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    ec2_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    ec2 = session.client("ec2", region_name=region)

    try:
        ec2.reboot_instances(InstanceIds=[ec2_id], DryRun=DRY_RUN)

    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
```

```
        )
        return

    response_action_message = f"Successfully rebooted {ec2_id}."
    response_action_status = constants.ResponseActionStatus.SUCCESS

    utils.send_response_action_result(
        presigned_url, scan_id, response_action_status,
response_action_message
```

AWS_EC2_RESPONSE_003 - Terminate compute instance

Terminates an EC2 instance for an incident response.
Mapped target entity type for generic response function: `virtualMachine`

AWS_EC2_RESPONSE_003

```
"""
Required permissions:
ec2:DescribeInstances
ec2:TerminateInstances
"""

import boto3
from botocore.exceptions import ClientError, WaiterError

from .. import constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    ec2_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]

    ec2_client = session.client("ec2", region_name=region)

    try:
        describe_response = ec2_client.describe_instances(InstanceIds=
[ec2_id])

    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
```

```
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    if (
        ("Reservations" not in describe_response)
        or (len(describe_response["Reservations"]) < 1)
        or ("Instances" not in describe_response["Reservations"][0])
    ):
        response_action_message = (
```

AWS_EC2_RESPONSE_004 - Take a snapshot of a compute instance

Takes a snapshot of each of the volumes of an ec2 instance and tags those snapshots
Mapped target entity type for generic response function: `virtualMachine`

AWS_EC2_RESPONSE_004

```
"""
Required permissions:
ec2:DescribeInstances
ec2:CreateSnapshot
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    ec2_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    ec2 = session.client("ec2", region_name=region)

    # get instance info (we need the volume id(s))
```

```
    try:
        result = ec2.describe_instances(InstanceIds=[ec2_id])

    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    if (
        ("Reservations" not in result)
        or (len(result["Reservations"]) < 1)
        or ("Instances" not in result["Reservations"][0])
    ):
```

AWS_EC2_RESPONSE_005 - Scale auto scaling group to 0

Changes MinSize, MaxSize and DesiredCapacity of an auto scaling group to 0 in order to down scale the group to 0
Mapped target entity type for generic response function: `autoScalingGroup`

AWS_EC2_RESPONSE_005

```
"""
Required permissions:
autoscaling:DescribeAutoScalingGroups
autoscaling:UpdateAutoScalingGroup
autoscaling:CreateOrUpdateTags
"""

import re
import time
from datetime import datetime, timedelta

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False
MAX_SCALE_DOWN_WAIT_MINUTES = 4

AUTO_SCALING_GROUP_REGEX = "arn:aws:autoscaling:[a-zA-Z0-9-]+:(?
P<account_id>\\d{12}):autoScalingGroup:[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-
fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}:autoScalingGroupName/(?
P<auto_scaling_group_name>.*)"
```

```
def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    auto_scaling_group_arn = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    re_match = re.match(AUTO_SCALING_GROUP_REGEX,
auto_scaling_group_arn)

    if re_match is None or len(re_match.groupdict()) != 2:
        response_action_message = f"external_id received:
{auto_scaling_group_arn} doesn't match auto scaling group arn format:
{AUTO_SCALING_GROUP_REGEX}"
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
```

AWS_EC2_RESPONSE_006 - Detach role from compute instance

Removes the specified IAM role from the specified Amazon EC2 instance profile
Mapped target entity type for generic response function: `virtualMachine`

AWS_EC2_RESPONSE_006

```
"""
Required permissions:
ec2:DescribeIamInstanceProfileAssociations
ec2:DisassociateIamInstanceProfile
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    ec2_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]
```

```
    ec2_client = session.client("ec2", region_name=region)

    # get the existing instance profile association ID
    print("Get existing instance profile config")
    try:
        instance_profile_association = (
            ec2_client.describe_iam_instance_profile_associations(
                Filters=[{"Name": "instance-id", "Values": [ec2_id]}]
            )
        )
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    if (
        ("IamInstanceProfileAssociations" not in
instance_profile_association)
        or
(len(instance_profile_association["IamInstanceProfileAssociations"]) <
```

AWS_EC2_RESPONSE_007 - Isolate EC2 instance from all networks

Changes the security group of the EC2 instance to a new security group which doesn't allow anything

Mapped target entity type for generic response function: `virtualMachine`

AWS_EC2_RESPONSE_007

```
"""
Required permissions:
ec2:RevokeSecurityGroupEgress
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils

WIZ_EMPTY_SECURITY_GROUP_NAME = "wiz-allow-nothing-sg"

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, event: dict, lambda_context):
```

```
    """
    Main Function invoked by index_parser.py
    """

    ec2_id = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    response_action_message, response_action_status =
isolate_ec2_instance(
        session, region, ec2_id
    )

    utils.send_response_action_result(
        presigned_url, scan_id, response_action_status,
response_action_message
    )

    if response_action_status == constants.ResponseActionStatus.SUCCESS:
        auto_tagging.autotag_ec2(
            ec2_id, region, subscription_id, presigned_url, scan_id
        )


def isolate_ec2_instance(session: boto3.Session, region: str, ec2_id:
str):
    ec2_client = session.client("ec2", region_name=region)
```

AWS_EC2_RESPONSE_008 - Detach EC2 or Lambda from all load balancers

Deregisters the EC2 instance or Lambda function from all target groups
Mapped target entity types for generic response function:

AWS_EC2_RESPONSE_008

```
"""
Required permissions:
elasticloadbalancing:DescribeTargetGroups
elasticloadbalancing:DescribeTargetHealth
elasticloadbalancing:DeregisterTargets
elasticloadbalancing:AddTags
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils
```

```
def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    resource_type = event["resource_type"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    if resource_type == "virtualMachine":
        target_type = "instance"
        resource_id = event["external_id"]
    elif resource_type == "lambda":
        target_type = "lambda"
        resource_id = event["external_id"]
    else:
        response_action_message = f"Invalid resource_type
{resource_type}"
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    response_action_message, response_action_status,
target_groups_deregistered = (
        detach_ec2_lambda(session, region, resource_id, target_type)
    )

    utils.send_response_action_result(
```

AWS_ECS_RESPONSE_001 - Stop ECS Service

Changes the desiredCount of an ECS Service to 0 in order to stop all tasks of the service
Mapped target entity type for generic response function: `ecs#service`

AWS_ECS_RESPONSE_001

```
"""
Required permissions:
ecs:UpdateService
ecs:DescribeServices
ecs:TagResource
"""

import re
```

```
import boto3
from botocore.exceptions import ClientError, WaiterError

from .. import auto_tagging, constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False

ECS_SERVICE_REGEX = "arn:aws:ecs:[a-zA-Z0-9-]+:(?
P<account_id>\\d{12}):service/(?P<cluster_name>[a-zA-Z0-9_-]+)/(?
P<ecs_service_name>[a-zA-Z0-9_-]+)"
SERVICE_STOP_WAIT_MINUTES = 4


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    ecs_service_arn = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    ecs_client = session.client("ecs", region_name=region)

    re_match = re.match(ECS_SERVICE_REGEX, ecs_service_arn)

    if re_match is None or len(re_match.groupdict()) != 3:
        response_action_message = f"external_id received:
{ecs_service_arn} doesn't match ECS service arn format:
{ECS_SERVICE_REGEX}"
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
```

AWS_ECS_RESPONSE_002 - Stop ECS Task

Stops an ECS Task. If the task is part of a service, this response action will fail. The ECS Task should be stopped by stopping the ECS Service via AWS_ECS_RESPONSE_001
Mapped target entity type for generic response function: `ecs#service`

AWS_ECS_RESPONSE_002

```
"""
Required permissions:
ecs:DescribeTasks
```

```
ecs:StopTask
ecs:TagResource
"""

import re

import boto3
from botocore.exceptions import ClientError, WaiterError

from .. import constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False

ECS_TASK_REGEX = "arn:aws:ecs:[a-zA-Z0-9-]+:(?
P<account_id>\\d{12}):task/(?P<cluster_name>[a-zA-Z0-9_-]+)/(?
P<ecs_task_name>[a-zA-Z0-9_-]+)"
SERVICE_STOP_WAIT_MINUTES = 4


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    ecs_task_arn = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]

    ecs_client = session.client("ecs", region_name=region)

    re_match = re.match(ECS_TASK_REGEX, ecs_task_arn)

    if re_match is None or len(re_match.groupdict()) != 3:
        response_action_message = f"external_id received: {ecs_task_arn}
doesn't match ECS task arn format: {ecs_task_arn}"
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return
```

AWS_IAM_001 - Enforce AWS account best practices password policy

Initiates stricter AWS account password policies by manipulating various settings in the password policy depending on the CCR
Mapped Cloud Configuration Rules: IAM-008, IAM-009, IAM-010, IAM-011, IAM-012, IAM-013, IAM-014, IAM-015

AWS_IAM_001

```
"""
Required Permissions:
- iam:UpdateAccountPasswordPolicy
"""

import boto3
from botocore.exceptions import ClientError

from .. import constants, utils

# password policy will only run ig dry run is set to False
dry_run = False


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]

    enforced_policy = {
        "MinimumPasswordLength": 14,
        "MaxPasswordAge": 90,
        "RequireLowercaseCharacters": True,
        "RequireUppercaseCharacters": True,
        "RequireNumbers": True,
        "RequireSymbols": True,
        "AllowUsersToChangePassword": True,
        "PasswordReusePrevention": 1,
    }

    # If there is an existing policy, set the restrictive
MinimumPasswordLength, PasswordReusePrevention, and MaxPasswordAge
    if "passwordPolicy" in event["metadata"]:
        print("Current password policy", event["metadata"]
["passwordPolicy"])
        current_policy = {
            k.lower(): v for k, v in event["metadata"]
["passwordPolicy"].items()
        }

        for key in ["MinimumPasswordLength", "PasswordReusePrevention"]:
            if (
                key.lower() in current_policy
                and current_policy[key.lower()] is not None
            ):
                enforced_policy[key] = max(
```

AWS_IAM_002 - Deactivate AWS access keys not used for 90 days or longer

Deactivates AWS access keys that have not been used for 90 days or more
Mapped Cloud Configuration Rule: IAM-045

AWS_IAM_002

```
"""
Required Permissions:
- iam:GetAccessKeyLastUsed
- iam:UpdateAccessKey
- iam:ListAccessKeys
"""

from datetime import date

import boto3
from botocore.exceptions import ClientError

from .. import constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    user_id = event["resource_name"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]

    iam = session.client("iam", region_name=region)

    # Get all access keys associated to a user
    try:
        iam_keys = iam.list_access_keys(UserName=user_id)
        access_keys = iam_keys["AccessKeyMetadata"]
        if access_keys:
            print("Access keys discovered : {0}".format(access_keys))
        else:
            print("No access keys discovered")
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return
```

```
    # Get last used date for each IAM access key
    for key in access_keys:
        try:
```

AWS_IAM_003 - IAM policy should not grant full administrative privileges to all AWS services

Removes the full admin policy and replaces it with a placeholder policy that has the minimum required access.
Mapped Cloud Configuration Rule: IAM-025

AWS_IAM_003

```
"""
Required permissions:
- iam:CreatePolicyVersion
- iam:PutRolePolicy
- iam:PutGroupPolicy
- iam:PutUserPolicy
- iam:TagPolicy
- iam:TagRole
- iam:TagUser
"""

import os

import boto3

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    # hacking the policy name into an ARN... since the providerId isn't
the same as externalId which gives the ARN.
    # resource_id = event['resource_id']
    resource_id = (
        "arn:aws:iam::"
        + event["subscription"]["id"]
        + ":policy/"
        + event["resource_name"]
    )

    policy_type = event["resource_type"]
    policy_external_id = event["external_id"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]
```

```
    region = event["region"]

    new_policy = '{\n    "Version": "2012-10-17",\n    "Statement": [\n
{\n            "Sid": "RemediatedAdminPolicy",\n            "Effect":
"Allow",\n            "Action": [\n
"sts:getCallerIdentity"\n            ],\n            "Resource": "*"\n
}\n    ]\n}'

    client = session.client("iam")
```

AWS_KMS_001 - Revokes pending deletion request for a KMS key

Removes the scheduled deletion of a KMS key. Data encrypted with a KMS key that is subsequently deleted will not be recoverable.
Mapped Cloud Configuration Rule: KMS-002

AWS_KMS_001

```
"""
Required Permissions:
- kms:CancelKeyDeletion
- kms:TagResource
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context,
params={}):
    """
    Main Function invoked by index_parser.py
    """
    print("Executing KMS delete key request playbook")

    key_id = event["external_id"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    region = event["region"]
    subscription_id = event["subscription"]["id"]

    client = session.client("kms")

    try:
        print("Removing deletion request for key : {0}".format(key_id))
        client.cancel_key_deletion(KeyId=key_id)
        response_action_message = "Key deletion request removed for key
ID : " + key_id
        response_action_status = constants.ResponseActionStatus.SUCCESS
        auto_tagging.autotag_kms_cmk(
```

```
            key_id.split("/")[1], region, subscription_id,
presigned_url, scan_id
        )
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )

    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        if "is not pending deletion" in response_action_message:
            response_action_message = (
```

AWS_LAMBDA_001 - Block public access to Lambda function

Removes statements where the Effect is 'allow' and Pricipal is * or AWS:* or s3.amazonaws.com and Condition is null or lambda:FunctionUrlAuthType:NONE
Mapped Cloud Configuration Rule: IAM-087

AWS_LAMBDA_001

```
"""
Required Permissions:
- lambda:GetPolicy
- lambda:RemovePermission
"""

import json

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    lambda_arn = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    lambda_client = session.client("lambda", region_name=region)

    try:
        lambda_policy_response =
```

```
lambda_client.get_policy(FunctionName=lambda_arn)
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    lambda_policy_str = lambda_policy_response.get("Policy")
    lambda_policy = json.loads(lambda_policy_str)
    statements = lambda_policy.get("Statement", [])
    removed_statements = []
    for statement in statements:
        if statement.get("Effect", "") != "Allow":
            continue
        principal = statement.get("Principal", "")
```

AWS_LAMBDA_RESPONSE_001 - Delete lambda function

Deletes a lambda function as part of an incident response
Mapped target entity type for generic response function: `lambda`

AWS_LAMBDA_RESPONSE_001

```
"""
Required permissions:
lambda:DeleteFunction
"""

import boto3
from botocore.exceptions import ClientError

from .. import constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    lambda_arn = event["external_id"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]

    lambda_client = session.client("lambda", region_name=region)
```

```
    try:
        lambda_client.delete_function(FunctionName=lambda_arn)
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    response_action_message = f"Successfully deleted function
{lambda_arn}."
    response_action_status = constants.ResponseActionStatus.SUCCESS

    utils.send_response_action_result(
        presigned_url, scan_id, response_action_status,
response_action_message
    )

    # note: cannot auto tag since the lambda is deleted so there is no
```

AWS_LAMBDA_RESPONSE_002 - Set function concurrency to 0

Sets a lambda function concurrency to 0, so it can't execute further
Mapped target entity type for generic response function: `lambda`

AWS_LAMBDA_RESPONSE_002

```
"""
Required permissions:
lambda:PutFunctionConcurrency
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils

# Options:
# Future support for passing DRY_RUN param
DRY_RUN = False


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    lambda_arn = event["external_id"]
    region = event["region"]
```

```
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    lambda_client = session.client("lambda", region_name=region)

    try:
        response = lambda_client.put_function_concurrency(
            FunctionName=lambda_arn, ReservedConcurrentExecutions=0
        )
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    response_check_zero = response["ReservedConcurrentExecutions"]
    if response_check_zero == 0:
        response_action_message = (
            f"Successfully set function {lambda_arn} concurrency to 0."
        )
        response_action_status = constants.ResponseActionStatus.SUCCESS
```

AWS_LAMBDA_RESPONSE_003 - Detach function resource policies

Detaches *all* resource policies from the Lambda function
Mapped target entity type for generic response function: `lambda`

AWS_LAMBDA_RESPONSE_003

```
"""
Required permissions:
lambda:GetPolicy
lambda:RemovePermission
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    lambda_arn = event["external_id"]
    region = event["region"]
```

```
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    lambda_client = session.client("lambda", region_name=region)

    # get policy statements
    try:
        lambda_policy =
lambda_client.get_policy(FunctionName=lambda_arn)
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        print(response_action_message)
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    if ("Policy" not in lambda_policy) or (lambda_policy["Policy"] ==
""):
        response_action_message = f'Lambda "{lambda_arn}" resource
policy already empty'
        response_action_status = constants.ResponseActionStatus.FAILURE
        print(response_action_message)
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
```

AWS_RDS_002 - Remove public access to RDS Database

Activates private access settings for RDS databases, ensuring they are only accessible within your VPC. This function verifies the PubliclyAccessible property of the database instance; if set to True, it modifies it to False to enhance security.
Mapped Cloud Configuration Rule: RDS-003

AWS_RDS_002

```
"""
Required Permissions:
- rds:DescribeDBInstances
- rds:ModifyDBInstance
- rds:AddTagsToResource
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils
```

```python
def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    resource_name = event["resource_name"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    external_id = event["metadata"]["externalId"]
    subscription_id = event["subscription"]["id"]

    rds = session.client("rds", region_name=region)

    try:
        db_instance = rds.describe_db_instances(
            Filters=[{"Name": "db-instance-id", "Values":
[resource_name]}]
        )["DBInstances"]

    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    try:
        public = db_instance[0]["PubliclyAccessible"]
    except (KeyError, IndexError) as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
```

AWS_REDSHIFT_001 - Remove public access to Redshift cluster

Verifies if the Redshift cluster has the PubliclyAccessible property activated. If set to True, it modifies it to False to enhance security.
Mapped Cloud Configuration Rule: [Redshift-002]

AWS_REDSHIFT_001

```python
"""
Required Permissions:
- redshift:DescribeClusters
- redshift:ModifyCluster
- redshift:CreateTags
"""

import boto3
from botocore.exceptions import ClientError
```

```
from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    cluster_id = event["resource_name"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    external_id = event["metadata"]["externalId"]
    subscription_id = event["subscription"]["id"]

    redshift = session.client("redshift", region_name=region)

    try:
        cluster =
redshift.describe_clusters(ClusterIdentifier=cluster_id)["Clusters"]
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
        return

    try:
        public = cluster[0]["PubliclyAccessible"]
    except (KeyError, IndexError) as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
```

AWS_S3_002 - Remove violating ACL from S3 bucket

Removes grants which violate the given CCR from an S3 bucket.
Mapped Cloud Configuration Rules: S3-003, S3-004, S3-005, S3-006, S3-007, S3-031, S3-033

AWS_S3_002

```
"""
Required Permissions:
- s3:GetBucketAcl
- s3:PutBucketAcl
```

```
- s3:PutBucketTagging
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils

VIOLATING_ACL_MAPPING = {
    "S3-003": {
        "URIs": [
            "http://acs.amazonaws.com/groups/global/AuthenticatedUsers",
            "http://acs.amazonaws.com/groups/global/AllUsers",
        ],
        "Permissions": ["FULL_CONTROL", "WRITE_ACP"],
    },
    "S3-004": {
        "URIs": [
            "http://acs.amazonaws.com/groups/global/AuthenticatedUsers",
            "http://acs.amazonaws.com/groups/global/AllUsers",
        ],
        "Permissions": ["FULL_CONTROL", "READ"],
    },
    "S3-005": {
        "URIs": [
            "http://acs.amazonaws.com/groups/global/AuthenticatedUsers",
            "http://acs.amazonaws.com/groups/global/AllUsers",
        ],
        "Permissions": ["FULL_CONTROL", "WRITE"],
    },
    "S3-006": {
        "URIs": [
            "http://acs.amazonaws.com/groups/global/AuthenticatedUsers",
            "http://acs.amazonaws.com/groups/global/AllUsers",
        ],
        "Permissions": ["FULL_CONTROL", "READ_ACP"],
    },
    "S3-007": {
        "URIs": [
            "http://acs.amazonaws.com/groups/global/AuthenticatedUsers",
            "http://acs.amazonaws.com/groups/global/AllUsers",
        ],
        "Permissions": [],
```

AWS_S3_003 - Enable object versioning for S3 bucket

Enhances data protection within an S3 bucket by enabling object versioning, a non-reversible process that ensures historical versions of objects are available for recovery.
Mapped Cloud Configuration Rule: S3-002

AWS_S3_003

```
"""
Required Permissions:
- s3:PutBucketVersioning
- s3:PutBucketTagging
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    bucket = event["resource_name"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    s3 = session.client("s3", region_name=region)

    try:
        s3.put_bucket_versioning(
            Bucket=bucket, VersioningConfiguration={"Status": "Enabled"}
        )
        response_action_status = constants.ResponseActionStatus.SUCCESS
        response_action_message = (
            "Object Versioning is enabled for S3 bucket : " + bucket
        )
        auto_tagging.autotag_s3_bucket(
            bucket, region, subscription_id, presigned_url, scan_id
        )
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )

    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        utils.send_response_action_result(
            presigned_url, scan_id, response_action_status,
response_action_message
        )
```

AWS_S3_004 - Enable S3 Bucket logging

Enables S3 bucket logging to obtain detailed insights into bucket activities. Though AWS doesn't enable this feature by default, this function verifies and enables the logging policy for the designated bucket, enhancing your monitoring capabilities. Mapped Cloud Configuration Rule: S3-001

AWS_S3_004

```
"""
Required Permissions:
- sts:GetCallerIdentity
- s3:CreateBucket
- s3:GetBucketLogging
- s3:PutBucketLogging
- s3:PutBucketTagging
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def remediate(session: boto3.Session, event: dict, lambda_context):
    """
    Main Function invoked by index_parser.py
    """

    bucket_name = event["resource_name"]
    region = event["region"]
    scan_id = event["scanId"]
    presigned_url = event["presignURL"]
    subscription_id = event["subscription"]["id"]

    s3 = session.client("s3", region_name=region)
    sts = session.client("sts", region_name=region)

    print("Checking current bucket logging configuration")
    try:
        logging = s3.get_bucket_logging(Bucket=bucket_name)
        read_logging_configuration = True
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        return

    # check if there's already logging enabled on the bucket
    if read_logging_configuration:
        try:
            enabled = logging["LoggingEnabled"]
        except KeyError:
            print("Logging not enabled on bucket
```

```
{0}".format(bucket_name))
            enabled = None

        if enabled is not None:
```

AWS_S3_005 - S3 Bucket should prohibit public read access

Removes any bucket policy statements which are too permissive (wildcard Principal) and set the public access block to True.
Mapped Cloud Configuration Rules: S3-046, S3-047

AWS_S3_005

```
"""
Required Permissions:
- s3:GetBucketPolicy
- s3:DeleteBucketPolicy
- s3:PutBucketPolicy
- s3:GetBucketPublicAccessBlock
- s3:PutBucketPublicAccessBlock
"""

import boto3
from botocore.exceptions import ClientError
import json

from .. import auto_tagging, constants, utils

from .AWS_S3_RESPONSE_001 import get_public_access_block,
set_public_access_block_true

ACTIONS_MAPPING = {
    "S3-046": ["*", "s3:*", "s3*", "s3:list", "s3:get"],
    "S3-047": ["*", "s3:*", "s3*", "s3:delete", "s3:put"],
}


def check_action(action: str, actions_to_check: list):
    for s3read_action in actions_to_check:
        if action.lower().startswith(s3read_action):
            return True
    return False


def check_actions(actions: list, actions_to_check: list):
    for action in actions:
        if check_action(action, actions_to_check):
            return True
    return False
```

```python
def check_principal_values(principal_values: list):
    for principal_value in principal_values:
        if isinstance(principal_value, str) and principal_value == "*":
            return True
        if isinstance(principal_value, list) and "*" in principal_value:
            return True
    return False
```

AWS_S3_RESPONSE_001 - Block bucket public access

Modifies the PublicAccessBlock configuration for an Amazon S3 bucket to restrict access: BlockPublicAcls, IgnorePublicAcls, BlockPublicPolicy and RestrictPublicBuckets are set to True
Mapped target entity type for generic response function: `bucket`

AWS_S3_RESPONSE_001

```python
"""
Required permissions:
s3:GetBucketPublicAccessBlock
s3:PutBucketPublicAccessBlock
"""

import boto3
from botocore.exceptions import ClientError

from .. import auto_tagging, constants, utils


def get_public_access_block(s3_client, s3_id):
    try:
        response = s3_client.get_public_access_block(Bucket=s3_id)
    except ClientError as e:
        response_action_message = e.response["Error"]["Message"]
        response_action_status = constants.ResponseActionStatus.FAILURE
        return response_action_message, response_action_status, None

    if "PublicAccessBlockConfiguration" in response:
        public_access_block_conf =
response["PublicAccessBlockConfiguration"]
        print(
            f"Current PublicAccessBlockConfiguration for the bucket:\n
{public_access_block_conf}"
        )
    else:
        response_action_message = (
            f"Got invalid response for get_public_access_block:
{response}"
        )
```

```
        response_action_status = constants.ResponseActionStatus.FAILURE
        return response_action_message, response_action_status, None

    return "", constants.ResponseActionStatus.SUCCESS,
public_access_block_conf


def set_public_access_block_true(s3_client, s3_id):
    try:
        s3_client.put_public_access_block(
            Bucket=s3_id,
            PublicAccessBlockConfiguration={
                "BlockPublicAcls": True,
                "IgnorePublicAcls": True,
                "BlockPublicPolicy": True
```

Updated 29 days ago

← Create Custom Response Functions

Troubleshoot Remediation & Response →

Did this page help you? Yes No