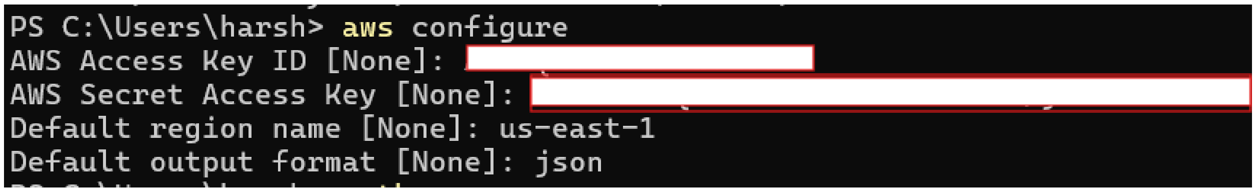
IAM Monitoring Module Implementation Documentation

Step 1: Create an AWS Account.

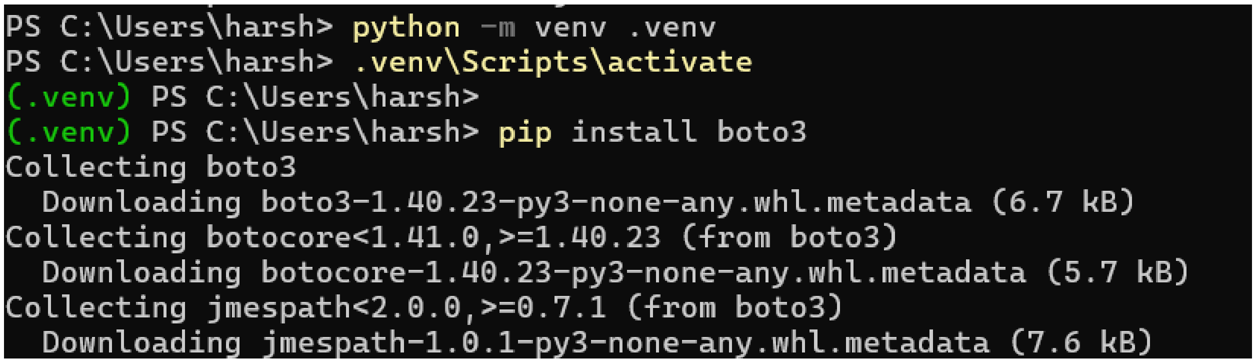
Step 2: Create a test **IAM User** in AWS for monitoring purpose and assign policies such as **SecurityAudit**. Also, add a policy which gives suspiciously excess permissions such as **IAMFullAccess.** This is done to simulate a condition where our program is triggered.

Step 3: Install **AWS CLI** for Windows and configure credentials such as **AWS Access Key, Secret Access Key**, and default region and output format.



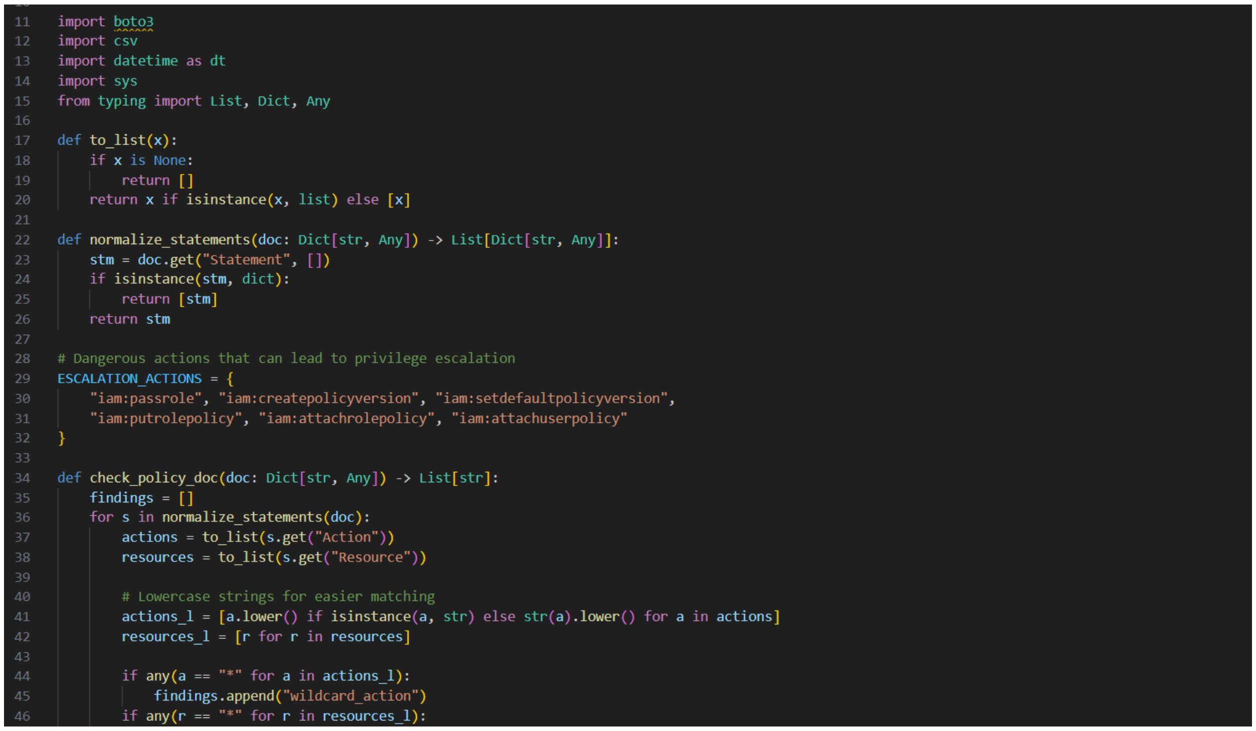
```
PS C:\Users\harsh> aws configure
AWS Access Key ID [None]: [redacted]
AWS Secret Access Key [None]: [redacted]
Default region name [None]: us-east-1
Default output format [None]: json
```

Step 4: Install Python and **boto3** (AWS SDK for Python).

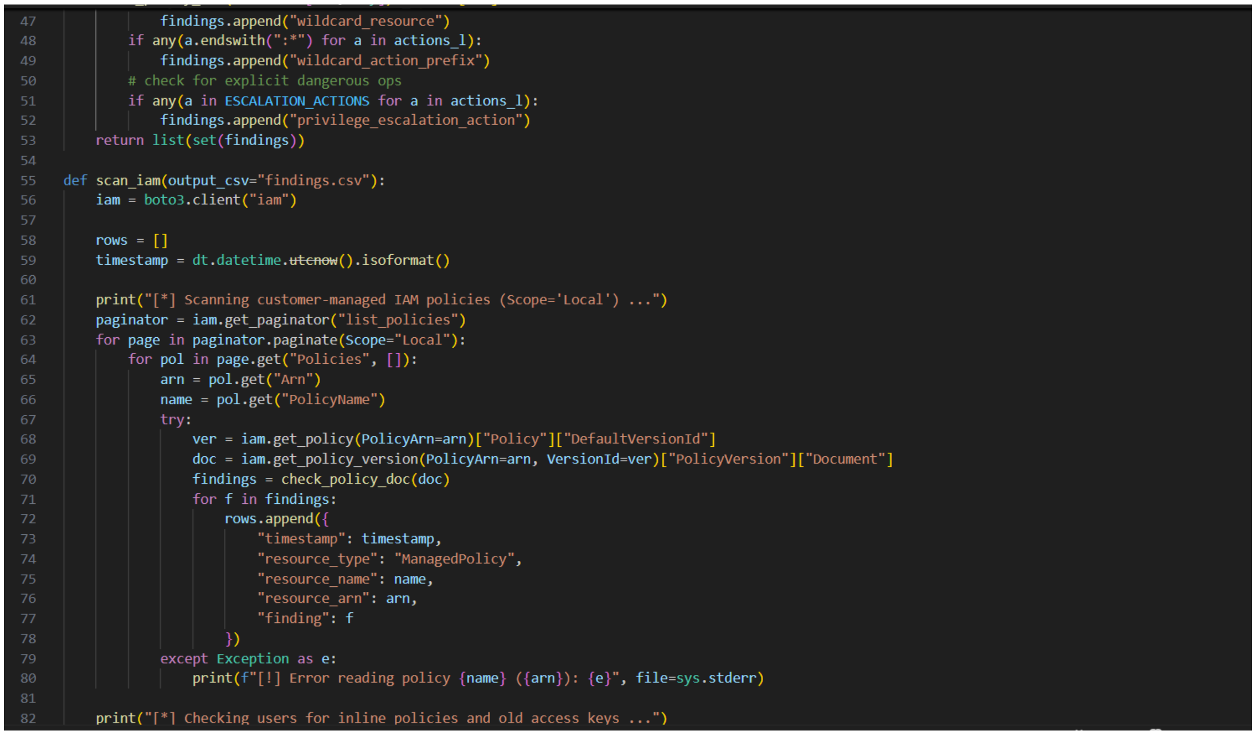


```
PS C:\Users\harsh> python -m venv .venv
PS C:\Users\harsh> .venv\Scripts\activate
(.venv) PS C:\Users\harsh>
(.venv) PS C:\Users\harsh> pip install boto3
Collecting boto3
  Downloading boto3-1.40.23-py3-none-any.whl.metadata (6.7 kB)
Collecting botocore<1.41.0,>=1.40.23 (from boto3)
  Downloading botocore-1.40.23-py3-none-any.whl.metadata (5.7 kB)
Collecting jmespath<2.0.0,>=0.7.1 (from boto3)
  Downloading jmespath-1.0.1-py3-none-any.whl.metadata (7.6 kB)
```

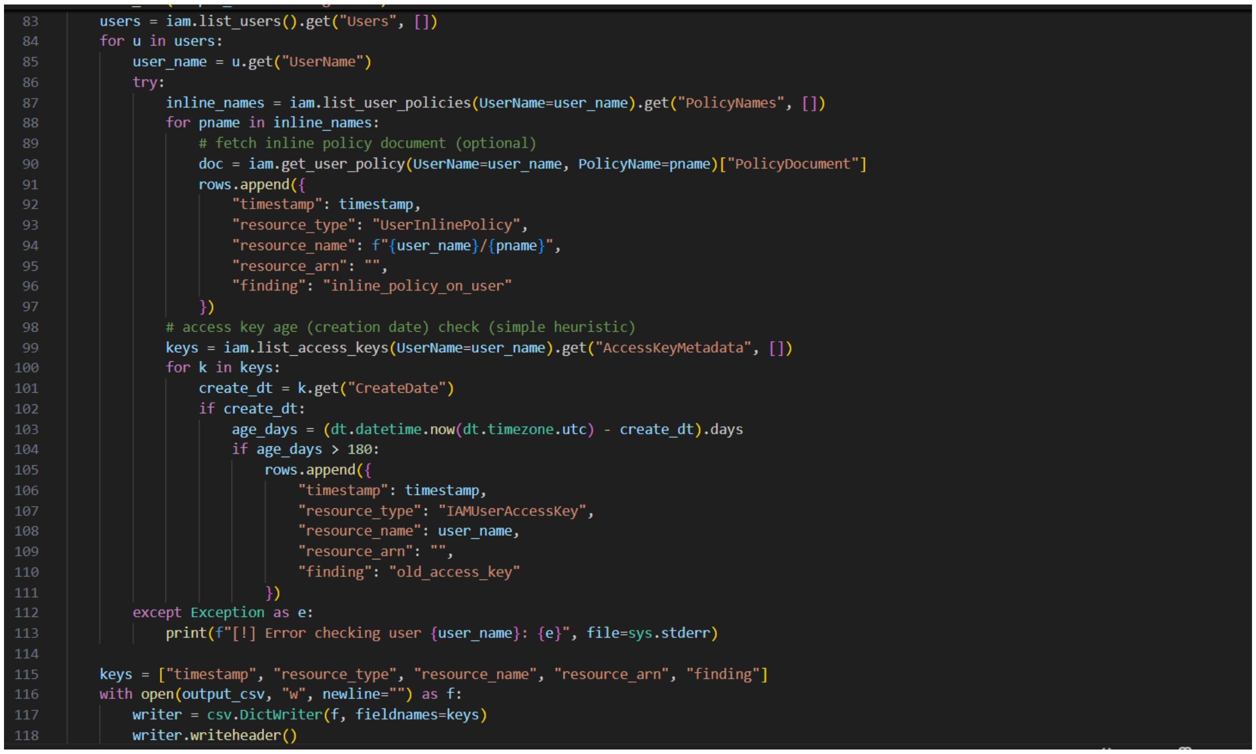
Step 5: Create a file named **iam_monitor.py** and write the following code:



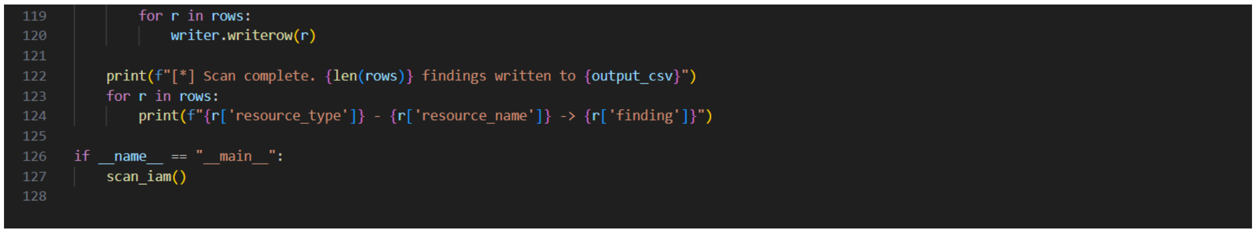
```python
import boto3
import csv
import datetime as dt
import sys
from typing import List, Dict, Any

def to_list(x):
    if x is None:
        return []
    return x if isinstance(x, list) else [x]

def normalize_statements(doc: Dict[str, Any]) -> List[Dict[str, Any]]:
    stm = doc.get("Statement", [])
    if isinstance(stm, dict):
        return [stm]
    return stm

# Dangerous actions that can lead to privilege escalation
ESCALATION_ACTIONS = {
    "iam:passrole", "iam:createpolicyversion", "iam:setdefaultpolicyversion",
    "iam:putrolepolicy", "iam:attachrolepolicy", "iam:attachuserpolicy"
}

def check_policy_doc(doc: Dict[str, Any]) -> List[str]:
    findings = []
    for s in normalize_statements(doc):
        actions = to_list(s.get("Action"))
        resources = to_list(s.get("Resource"))

        # Lowercase strings for easier matching
        actions_l = [a.lower() if isinstance(a, str) else str(a).lower() for a in actions]
        resources_l = [r for r in resources]

        if any(a == "*" for a in actions_l):
            findings.append("wildcard_action")
        if any(r == "*" for r in resources_l):
```



```python
            findings.append("wildcard_resource")
        if any(a.endswith(":*") for a in actions_l):
            findings.append("wildcard_action_prefix")
        # check for explicit dangerous ops
        if any(a in ESCALATION_ACTIONS for a in actions_l):
            findings.append("privilege_escalation_action")
    return list(set(findings))

def scan_iam(output_csv="findings.csv"):
    iam = boto3.client("iam")

    rows = []
    timestamp = dt.datetime.utcnow().isoformat()

    print("[*] Scanning customer-managed IAM policies (Scope='Local') ...")
    paginator = iam.get_paginator("list_policies")
    for page in paginator.paginate(Scope="Local"):
        for pol in page.get("Policies", []):
            arn = pol.get("Arn")
            name = pol.get("PolicyName")
            try:
                ver = iam.get_policy(PolicyArn=arn)["Policy"]["DefaultVersionId"]
                doc = iam.get_policy_version(PolicyArn=arn, VersionId=ver)["PolicyVersion"]["Document"]
                findings = check_policy_doc(doc)
                for f in findings:
                    rows.append({
                        "timestamp": timestamp,
                        "resource_type": "ManagedPolicy",
                        "resource_name": name,
                        "resource_arn": arn,
                        "finding": f
                    })
            except Exception as e:
                print(f"[!] Error reading policy {name} ({arn}): {e}", file=sys.stderr)

    print("[*] Checking users for inline policies and old access keys ...")
```

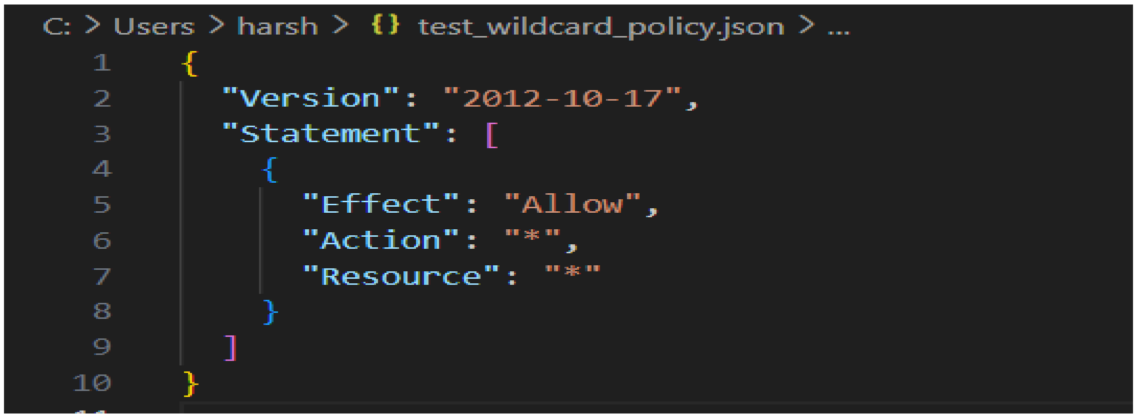


```
    users = iam.list_users().get("Users", [])
    for u in users:
        user_name = u.get("UserName")
        try:
            inline_names = iam.list_user_policies(UserName=user_name).get("PolicyNames", [])
            for pname in inline_names:
                # fetch inline policy document (optional)
                doc = iam.get_user_policy(UserName=user_name, PolicyName=pname)["PolicyDocument"]
                rows.append({
                    "timestamp": timestamp,
                    "resource_type": "UserInlinePolicy",
                    "resource_name": f"{user_name}/{pname}",
                    "resource_arn": "",
                    "finding": "inline_policy_on_user"
                })
            # access key age (creation date) check (simple heuristic)
            keys = iam.list_access_keys(UserName=user_name).get("AccessKeyMetadata", [])
            for k in keys:
                create_dt = k.get("CreateDate")
                if create_dt:
                    age_days = (dt.datetime.now(dt.timezone.utc) - create_dt).days
                    if age_days > 180:
                        rows.append({
                            "timestamp": timestamp,
                            "resource_type": "IAMUserAccessKey",
                            "resource_name": user_name,
                            "resource_arn": "",
                            "finding": "old_access_key"
                        })
        except Exception as e:
            print(f"[!] Error checking user {user_name}: {e}", file=sys.stderr)

    keys = ["timestamp", "resource_type", "resource_name", "resource_arn", "finding"]
    with open(output_csv, "w", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=keys)
        writer.writeheader()
```



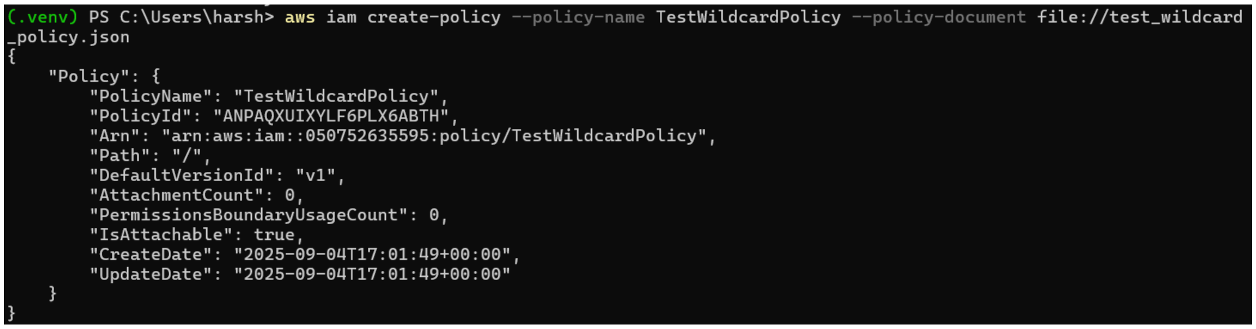
```
        for r in rows:
            writer.writerow(r)

    print(f"[*] Scan complete. {len(rows)} findings written to {output_csv}")
    for r in rows:
        print(f"{r['resource_type']} - {r['resource_name']} -> {r['finding']}")

if __name__ == "__main__":
    scan_iam()
```

Step 6: Use AWS CLI to create a **customer-managed policy** with **wildcards** so the script flags as possible IAM least privilege violation.



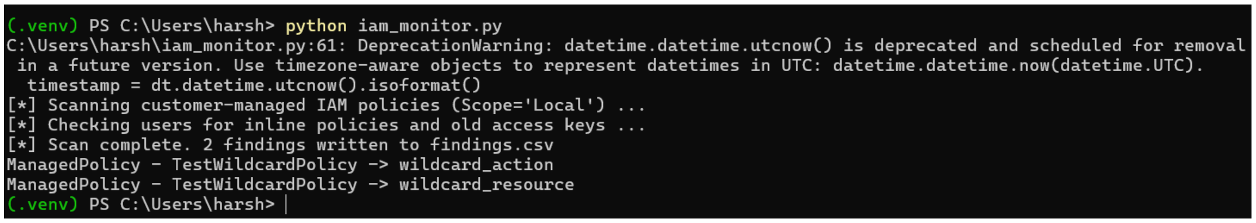
```
C: > Users > harsh > {} test_wildcard_policy.json > ...
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
```

Step 7: This command **creates a dangerously permissive IAM policy** named TestWildcardPolicy, so that your monitoring tool has something to detect and report.



```
(.venv) PS C:\Users\harsh> aws iam create-policy --policy-name TestWildcardPolicy --policy-document file://test_wildcard
_policy.json
{
    "Policy": {
        "PolicyName": "TestWildcardPolicy",
        "PolicyId": "ANPAQXUIXYLF6PLX6ABTH",
        "Arn": "arn:aws:iam::050752635595:policy/TestWildcardPolicy",
        "Path": "/",
        "DefaultVersionId": "v1",
        "AttachmentCount": 0,
        "PermissionsBoundaryUsageCount": 0,
        "IsAttachable": true,
        "CreateDate": "2025-09-04T17:01:49+00:00",
        "UpdateDate": "2025-09-04T17:01:49+00:00"
    }
}
```

Step 8: Run the script and it will flag for bad policies write its findings to findings.csv

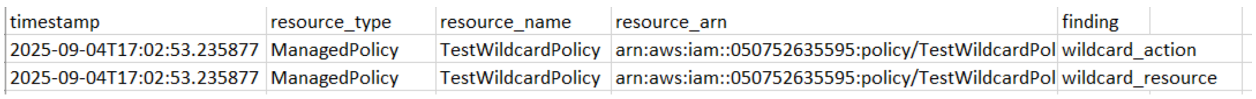


```
(.venv) PS C:\Users\harsh> python iam_monitor.py
C:\Users\harsh\iam_monitor.py:61: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled for removal
 in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
  timestamp = dt.datetime.utcnow().isoformat()
[*] Scanning customer-managed IAM policies (Scope='Local') ...
[*] Checking users for inline policies and old access keys ...
[*] Scan complete. 2 findings written to findings.csv
ManagedPolicy - TestWildcardPolicy -> wildcard_action
ManagedPolicy - TestWildcardPolicy -> wildcard_resource
(.venv) PS C:\Users\harsh> |
```

- **ManagedPolicy** - TestWildcardPolicy -> wildcard_action
  → Your script detected that the policy allows Action: "*" (all actions).
- **ManagedPolicy** - TestWildcardPolicy -> wildcard_resource
  → It also flagged that the policy applies to Resource: "*", meaning it grants access to *any* resource.
- 2 findings written to findings.csv
  → The script stored the results in a CSV file, which you can

open in Excel, Notepad, or import into dashboards (e.g. Grafana) for visualization.

Findings.csv will document the following information regarding the policy creation.



timestamp	resource_type	resource_name	resource_arn	finding
2025-09-04T17:02:53.235877	ManagedPolicy	TestWildcardPolicy	arn:aws:iam::050752635595:policy/TestWildcardPol	wildcard_action
2025-09-04T17:02:53.235877	ManagedPolicy	TestWildcardPolicy	arn:aws:iam::050752635595:policy/TestWildcardPol	wildcard_resource

This module can be extended in future for functionalities such as:

Privilege escalation patterns (e.g., iam:PassRole, sts:AssumeRole, iam:CreatePolicyVersion).

Over-permissive S3 bucket policies (public Principal: "*") → check via s3.get_bucket_policy.

Unused IAM users/roles → flag accounts with no login or API activity for X days.

MFA enforcement → detect IAM users without MFA enabled.

Inactive access keys → disable or flag keys unused for > 90 days.

Root account usage → alert if root account used (best practice: avoid).