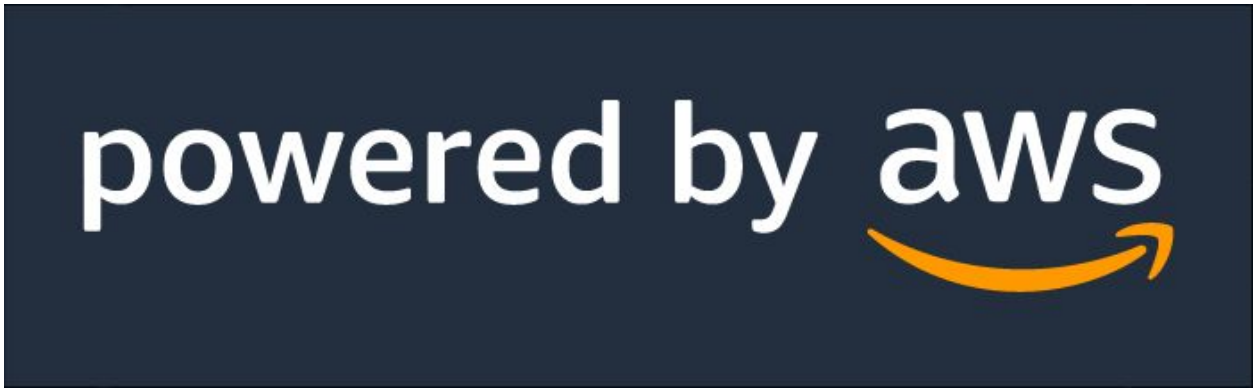


AWS Security Cheat Sheet

Created by Gekk05

INTRODUCTION

AWS is the Cloud computing forerunner of the modern world. Every year over a thousand new services are created in-house at AWS. I am a professional penetration tester and security engineer with experience in auditing AWS service accounts for security flaws and violations of best practices. This cheat sheet contains some of the most commonly used AWS services and their respective security flaws.

PURPOSE

This document was created in an effort to catalyze the AWS learning process for newly hired internal engineers with no prior AWS experience. AWS is a beast, it's possible to know everything. The goal of this cheat sheet is to have an easy document to reference when auditing AWS account configurations for personal projects, contracted engagements, or whatever it may be. As of now, this cheat sheet will serve as a "live" document and is subject to change. I have plans to update this cheat sheet as new vulnerabilities are discovered and new services are introduced. Each AWS service contains a subset cheat sheet for easy reference.

AWS SERVICES

IAM

AWS IAM is an identity and management system for resources and services which allows fine-grained access control settings. IAM consists of users, groups, roles, and policies. A single AWS account often has multiple IAM users. To allocate permissions to specific users, IAM roles can be attached to them. Roles are essentially titles or labels, such as "admin". To assign permissions to the roles, you must attach an IAM policy. IAM policies are typically JSON blobs that specify who can access what.

Example:
IAM User: Joe
IAM Role: Administrator (simply a label or title)

IAM Policy: AdministratorAccess (AWS Managed policy)

AdministratorAccess Policy Example:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "*",
            "Resource": "*"
        }
    ]
}
```

IAM Policies

There are two primary types of IAM policies:

A. Managed policies - Created by Amazon, every AWS account has these. Managed policies are designed for common use cases, such as 'AdministratorAccess'.

B. Inline policies - Custom policies created by the AWS account owner.

Permission Boundaries

Specifying permission boundaries is done via the 'Statement' key.

The example policy below allows the DynamoDB action DescribeTable on ANY resource, denoted by the wildcard.

```
{
  "Statement": {
    "Effect": "Allow",
    "Action": "dynamodb:DescribeTable",
```

```
    "Resource": "*"
  }
}
```

The resource is typically an ARN, such as `arn:aws:dynamodb:<region>:<account-id>:<table-name>`

To scope this policy down to abide by the principle of least privilege, you should specify the resource value as the exact table being used. In the policy below we are allowing a specific action in a specific account on a specific table. It's also possible to allow multiple operations via specifying them, or wildcarding.

```
{
  "Statement": {
    "Effect": "Allow",
    "Action": "dynamodb:DescribeTable",
    "Resource":
"arn:aws:dynamodbo:us-east-1:01234568935:myTable"
  }
}
```

Resource Polices Versus Identity Policies

IAM policies can also be defined as "resource policies" or "identity policies".

1. Identity policies are attached to a user or a group of users
2. Resource policies are attached to services. Not all resources allow resource-based policy.

The key difference between the two is the usage of the "Principal" key. A principal is typically the ARN of a user or another service. Principals typically are not specified in Identity-based IAM policies as identity policies are attached directly to a principal. Resource policies are attached to a resource, then you specify what principals can access that resource.

Amazon S3 allows resource policies, below is an example where the policy allows any user in the AWS account 01234568935 to perform any action on the specified S3 bucket:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::01234568935:user/*"
      },
      "Action": "s3:*",
      "Resource": "arn:aws:s3:::<bucket-name>"
    }
  ]
}
```

Resource: AWS Services That Work with IAM

IAM Policy Best Practices And Evaluation

All IAM policies should abide by the principle of least privilege. Evaluating IAM policies can be difficult depending on the service complexity and the amount of moving parts in the service account. To adequately scope the IAM policy down as much as possible, the pentester must be familiar enough with the service to determine the bare minimum permissions needed for the service to operate.

Dangers Of Wildcarding

Very rarely is it necessary to wildcard resources or principals in IAM policies. Most likely this is a violation of the principle of least privilege and should be verified.

Example of an extremely dangerous case of wildcarding

```
{
  "Statement": {
```

```
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "arn:aws:iam::<account-id>:role/*"
  }
}
```

Example of a misconfigured principal

It's common for traffic to be routed via API gateway to invoke lambda functions. In the example below we're using a resource policy that allows apigateway.amazonaws.com to call the lambda function specified in the resource. Specifying "apigateway.amazonaws.com" as the principal allows anyone using API gateway to call the function. Because of this, any user who knows the lambda function ARN can create their own API with API gateway then call the Lambda function.

```
{
   "Action": "lambda:InvokeFunction",
   "Effect": "Allow",
   "Principal": {
          "Service": "apigateway.amazonaws.com"
          },
          "Resource":
"arn:aws:lambda:us-east-1:<account-id>:function:<function-name>"
}
```

Inherently Dangerous Policies

Some policies are inherently dangerous and can lead to privilege escalation

Exploiting Policy Modification Operations

These operations may allow an attacker to modify a policy and possibly attach policies with higher privileges:

01. iam:PutGroupPolicy
02. iam:PutRolePolicy
03. iam:PutUserPolicy
04. iam:AttachGroupPolicy
05. iam:AttachRolePolicy
06. iam:AttachUserPolicy

Exploiting Policy Versioning And Group Operations

Policies can have multiple versions. With these operations an attacker can create a new version with higher privileges and attach the policy to the role:

01. iam:CreatePolicyVersion
02. iam:SetDefaultPolicyVersion

An attacker can use this to add themselves to a group with higher privileges:

01. iam:AddUserToGroup

Simple Storage Service (AWS S3)

AWS S3 is a storage service commonly used to store logs, backups, static assets, and more. S3 is composed of buckets and objects which similarly resemble the directory-file relationship in most filesystems . S3 Buckets house S3 objects. A bucket can be perceived as a directory and an object can be seen as a file. To learn more about what should and shouldn't be classified as a finding, visit here: Technical Guides - Known Issues

S3 Security Checklist

01. **Server-side Encryption**

    S3 buckets containing sensitive information must have server-side encryption enabled. The best way to encrypt buckets is to enable default encryption at the bucket

level this way all objects stored in the bucket are also encrypted. An alternative is specifying encryption criteria when creating objects inside buckets. It's possible for a bucket to be unencrypted but with all of the objects encrypted at the object level. It's also possible to have some objects encrypted, but not all. This can happen when objects are created without encryption then a customer enables default encryption at the bucket level, thereby encrypting all new objects. It's important to check encryption at both the bucket level and the object level. Because S3 is commonly used to host web assets which are already publicly available and not considered sensitive, buckets used for such do not need to be encrypted.

Report template: S3 Buckets Without Server Side Encryption Enabled

BASH one-liner to list all S3 buckets with server-side encryption disabled:

```
for bucket in $(aws s3 ls | cut -d " " -f 3); do if [[ $(aws
s3api get-bucket-encryption --bucket $bucket 2>&1| grep -io
error | wc -c) -gt 0 ]]; then echo -e $bucket 'Unencrypted
bucket!';fi; done
```

02. Access Logging

Enabling access logging logs API calls pertaining to S3 buckets. Access logs are extremely useful during security audits and the logs enforce accountability, as you know who did what.

BASH one-liner to list all S3 buckets with access logging disabled:

```
for bucket in $(aws s3 ls | cut -d " " -f 3); do if [[ $(aws
s3api get-bucket-logging --bucket $bucket 2>&1 | wc -c) -eq 0
]];then echo -e $bucket '[Logging disabled]';fi; done
```

03. Versioning

Bucket versioning is an availability tool. With versioning enabled accidental object deletions can easily be recovered. Disabling versioning doesn't have direct security implications, but enabling versioning is considered a best practice when dealing with

important information.

04. Over-permissive IAM Policies

"Public" S3 buckets have historically haunted companies. Somehow S3 buckets containing sensitive information end up public. By default all S3 buckets block public access but sometimes developers programmatically write to buckets they forgot they had made public later on down the road. For a bucket to be considered public, there must be a degree of wildcarding in the S3 bucket's IAM (resource) policy; this is also known as "anonymous access".

See: IAM Policy Principle Of Least Privilege Violation

Example:
Anyone can perform the GetObject operation on this S3 bucket.

```
"Statement":[
   {
   "Effect":"Allow",
   "Principal": "*",
   "Action":["s3:GetObject"],
   "Resource":["arn:aws:s3:::examplebucket/*"]
   }
```

Not all cases of over-permissive IAM policies are as catastrophic as totally public buckets, this is a finding too

Example:
Here we're allowing any EC2 instance perform the GetObject operation on our bucket. While this isn't terrible, it violates the principle of least privilege and should be scoped down more.

```
"Statement":[
   {
```

```
"Effect":"Allow",
"Principal": "*",
"Action":["s3:GetObject"],
"Resource":["arn:aws:s3:::examplebucket/*"]
}
```

05. Bucket Sniping

S3 bucket names are globally unique which is why S3 ARNs don't need a region specified. When a bucket is deleted, the name is freed and anyone is open to register that exact bucket name. It's very common for services to write to S3 buckets programmatically. A problem arises when a service or application writes to an S3 bucket without verifying ownership. This is an issue because in a case where a developer deletes a bucket without taking into account the services automatically configured to write to it, an attacker can register the developer's old bucket name for themself and the developer's service would ignorantly continue to write data to a bucket under someone else's control. Bucket sniping can also lead to subdomain takeovers, cross-site scripting, and other flaws depending on the context the bucket is being used. If a service doesn't verify ownership of the bucket before writing to it, this should be reported.

Example:
Developers sometimes use S3 buckets to host static web pages on subdomains. If during subdomain brute forcing an attacker discovers the subdomain and notices the 404 bucket not found, recognizing this is a signal of bucket deletion, the attacker can re-register that S3 bucket and completely hijack the subdomain.

Note: AWS has implemented a buffer period where the previously deleted S3 bucket name cannot be re-registered within 24 hours. To learn more about bucket sniping and mitigations, read here: Bucket Sniping Protection

DynamoDB (AWS DDB)

DynamoDB is a fully managed NoSQL key-value database service and one of the most frequently used database management systems for AWS developers.

DynamoDB Security Checklist

01. Point-in-time Recovery

Disabling PITR doesn't directly come with security implications, but it's a best practice to have it enabled as it's an availability feature. PITR automatically creates snapchat backups of the database.

02. NoSQL Injection

Typically AWS developers use previously pentested DAOs and ORMs when dealing with DynamoD; in the select few cases when this isn't true, code review should include checks for injection vulnerabilities such as NoSQL injection tautology auth bypass with DynamoDB comparison operators.

Example:
Using DynamoDB's comparison operators, this query should always evaluate to true if the stars align and the queries handling log-in operations are vulnerable.

```
{"username": {"$ne": null}, "password": {"$ne": null}}
```

03. Input Validation

While input validation has nothing to do with DynamoDB, the service team should implement restrictions on data being stored in the database, such as length. DynamoDB has a 400 KB max item size. In general, you don't want arbitrary values being stored in the database.

AWS Key Management System (AWS KMS)

KMS is used to create and manage encryption keys that are used to encrypt data and CMKs (Customer Managed Keys) for envelope encryption. All key changes are logged in AWS CloudTrail.

KMS Security Checklist

01. Key Rotation

Rotating encryption keys is a security best practice. AWS managed keys are automatically rotated, but CMKs need key rotation explicitly enabled. AWS managed CMKs require annual automatic key rotation.

02. Over-permissive Key Policy

Like any AWS service, key policies can be over-permissive too.

Example:
This is a key's IAM (resource) policy which allows ANY user or service in ANY account to perform operations on all resources.

```
{
     "Sid": "Allow use of the key",
     "Effect": "Allow",
     "Principal": {"AWS": "*"},
     "Action": [
     "kms:DescribeKey",
     "kms:GetPublicKey",
     "kms:Sign",
     "kms:Verify"
     ],
     "Resource": "*"
```

```
}
```

See: IAM Policy Principle Of Least Privilege Violation

AWS CloudTrail

CloudTrail is another AWS logging and monitoring service. CloudTrail is primarily used for logging API calls and AWS account events, rather than service logs like CloudWatch does. CloudTrail logs are stored in S3 buckets.

AWS CloudTrail Security Checklist

01. **Verify CloudTrail is enabled**

02. **File Integrity Validation**

AWS CloudWatch

CloudWatch is the primary logging and monitoring service for all other AWS services. CloudWatch is composed of log groups and log streams. Typically each service is assigned its own log group. Log groups contain log streams, where the logs are stored.

AWS Cloudwatch Security Checklist

01. **Ensure all services being used have CloudWatch logging enabled**

02. **Logging violations**

Proper logging isn't the easiest thing to accomplish. More often than not there are logging violations. When evaluating logging, ask yourself:

a. Is everything being logged that needs to be logged for accountability and debugging purposes? If not, report it.
b. Is customer data exposed in log streams? Report it.
c. Is session information (Authentication cookies, passwords, secret keys) exposed in log streams? Report it.

Note on session information: oftentimes SigV4 tokens are logged. This is not an issue.

For more information pertaining to logging best practices, read here: Secure Logging Guidelines

Simple Notification Service (SNS)

AWS Simple Notification Service (SNS) is a fully managed notifications service used to fan out messages to a large number of subscriber endpoints for parallel processing. Subscribers can be in the form of people (SMS or Email) or other AWS services.

AWS SNS is comprised of 3 main parts:

01. Topics: A logical access point and communication channel
02. Subscribers: The endpoints which will receive the messages
03. Publisher: Who or what that is performing the "publish" action. The publish action simply sends a message to an SNS topic.

AWS SNS common use-case example:

Topic: snsLambdaTopic
Subscriber: AWS Lambda function
Publisher: You

AWS Lambda has an option to add triggers, such as SNS messages. When an SNS topic is added as an AWS Lambda trigger, it subscribes to the SNS topic. Once the Lambda function is

subscribed to the SNS topic, any messages published to the topic will be received by one of the Lambda function's input parameters.

SNS Security Checklist

01. Server-side encryption (SSE)

SNS topics should have server-side encryption enabled so when SNS messages are published to an SNS topic, the message is immediately encrypted.

Finding template: SNS Topic Without Server Side Encryption Enabled

BASH script one-liner to list all SNS topics in the AWS account with SSE disabled:

```
for topics in $(aws sns list-topics --region us-east-1 | grep -A 0 "TopicArn" | cut -d '"' -f 4); do for attributes in $(aws sns get-topic-attributes --topic $topics --region us-east-1 | grep "KmsMasterKeyId" | tr -d " " | wc -c); do if [[ $attributes -eq 0 ]]; then echo -e $topics '[Encryption Disabled]'; else echo -e $topics '[Encryption enabled]' ;fi; done; done;
```

02. Over-permissive IAM Policy

Manually verify each SNS topic's IAM policy. Wildcarded principles are typically very bad.

Example 1: This SNS topic would allow anyone to publish messages to it. Depending on who and what are subscribed to the SNS topic, this could possibly even compromise other services.

```
{
    "Effect": "Allow",
    "Principal": {
```

```
     "AWS": "*"
     },
     "Action": "SNS:Publish",
     "Resource":
"arn:aws:sns:us-east-1:012345678901:SNStopicName"
     },
```

Example 2: This SNS topic would allow anyone to subscribe to the SNS topic. Depending on the nature of the messages, this could disclose sensitive information.

```
{
     "Sid": "__console_sub_0",
     "Effect": "Allow",
     "Principal": {
     "AWS": "*"
     },
     "Action": [
     "SNS:Subscribe",
     "SNS:Receive"
     ],
     "Resource":
"arn:aws:sns:us-east-1:123456789012:cc-web-app-topic"
     },
```

AWS Elastic Compute Cloud (EC2)

AWS EC2 is a service that enables customers to deploy servers on the fly. EC2 instances are simply cloud servers, similar to VPS's. Each server is referred to as an "instance". EC2 instances can be used for anything, but are commonly used to host web servers. Like other services, permissions can be attached to EC2 instances. This is accomplished through EC2 instance profile roles.

EC2 Security Checklist

01. Over-permissive IAM Policy (Instance profile)

The EC2 instance profile may have a role containing an over-permissive policy. Similarly to how you wouldn't want your webserver running with excessive permissions, you don't want the EC2 instance to have excessive permissions. Because EC2 instances very often communicate with other AWS services, compromising the EC2 instance can easily lead to compromise of other services in the AWS account. A common example of this is the classic SSRF attack where an attacker will find an SSRF vulnerability in a web application hosted by an EC2 instance then query the internal metadata service to obtain credentials. Whatever permissions were assigned to the EC2 instance role, the attacker now how via SSRF. If the profile role has an admin policy attached to it, the attacker now has full control of the AWS account.

02. IMDSv1

Classic SSRF attacks leverage the internal metadata service of EC2 instances. The issue became so common that AWS introduced IMDSv2 which employs natural SSRF defenses such as a preliminary PUT request requiring security tokens. By default both IMDSv2 and IMDSv1 are enabled, however, customers need to explicitly enable token enforcement else the tokens are never validated, rendering IMDSv2 useless. It's still not very common for service teams to have IMDSv2 enforced as it can cause integration issues, but it's a good thing to talk to the service team about enabling.

https://aws.amazon.com/blogs/security/defense-in-depth-open-firewalls-reverse-proxies-ssrf-vulnerabilities-ec2-instance-metadata-service/

03. Over-permissive security groups

Each EC2 instance has its own security group. Security groups are essentially virtual firewalls. By default, when EC2 instances are created, the security group allows all inbound SSH traffic. Production instances shouldn't use default security groups, but similarly to how IAM policies are evaluated, manually verify the security group rules

(inbound and outbound) are scoped down as much as possible.

See: Security Group Policy Too Broad

04. EBS Server-Side Encryption

An Amazon EBS volume is a durable, block-level storage device that you can attach to EC2 instances -- they operate similarly to hard drives. If EBS volumes are being used, ensure server-side encryption is enabled.

BASH one-liner to find EBS volumes with SSE disabled:

```
for volumes in $(aws ec2 describe-volumes --region us-east-1|
grep -i "encrypted" | cut -d '"' -f 3 | tr -d ':, ' ); do if [[
"$volumes" = "true" ]]; then echo $volumes '[Not Encrypted]';
else echo $volumnes '[Encrypted]';fi;done;
```

See: EBS Volumes Not Encrypted

AWS API Gateway

API Gateway is a service that creates, maintains, and secures APIs. Often, API Gateway sits between the client and internal APIs. Requests are sent to API Gateway via the client, validated, parsed, then forwarded accordingly. Security is often implemented at the API Gateway level, such as content encoding, AWS WAF, request throttling, and more.

API Gateway security checklist

01. Request throttling configured

API Gateway utilizes the token bucket algorithm to throttle requests. Request throttling should be configured as a defense in depth measure.

02. **HTTP Request validation enabled**

03. **HTTP Methods enforced**

04. **CloudWatch logs enabled**

05. **Encryption in transit**
    Note: TLS termination is done at the API gateway level

AWS Lambda

AWS Lambda is a serverless event-driven computing service. AWS Lambda has many benefits such at automatic scaling and pay-per-request. When an AWS Lambda is invoked by a user or via an event trigger a temporary containerized environment is allocated to execute the Lambda's code. The file system is read-only, and writing to it wouldn't do much good as the environment is terminated upon execution completion.

Exploiting temporary Lambda environments requires some background knowledge about the container allocation process. As mentioned above, a huge benefit of AWS Lambda is the ability to pay-per-use. Before Lambda functions are executed, the environment is allocated CPU capacity and RAM. To calculate usage, AWS multiplies the original amount of memory allocated prior to Lambda execution against the total uptime of the Lambda. In order to save both AWS and the customer money and increase performance, environments are cached across different executions. The /tmp directory is a RAM disk, which is being cached and can be used for further exploitation as it can be written to. Lambda environments terminate after 4 minutes and 30 seconds, but to circumvent this we can flow requests to the lambda function, ensuring what we need remains in memory.

AWS Lambda Security Checklist

01. **OWASP Top 10**
    Like any other application, Lambda functions should be reviewed for common vulnerabilities.

02. **Hardcoded credentials**

Lambda functions should not contain hardcoded credentials of any kind

03. **Logging violations**

Lambda functions often cause logging violations. In AWS Lamba, any print() calls log the stdout to AWS CloudWatch. This often leads to logging of sensitive information, usually from HTTP requests.

Report template: Secure Logging Violation

04. **Over-permissive IAM Policy (Lambda execution role)**

AWS Lambda IAM policies need to be reviewed like any other service. Typically customers should not directly interact with Lambda functions. Instead, requests are filtered by AWS API Gateway then routed to AWS Lambdas, as AWS API Gateway provides more granular access controls and request validation. AWS Lambdas also should not be globally callable.

Example: Here we are allowing any prinicpal (any AWS user and AWS service) to invoke the "HelloWorld" Lambda function.

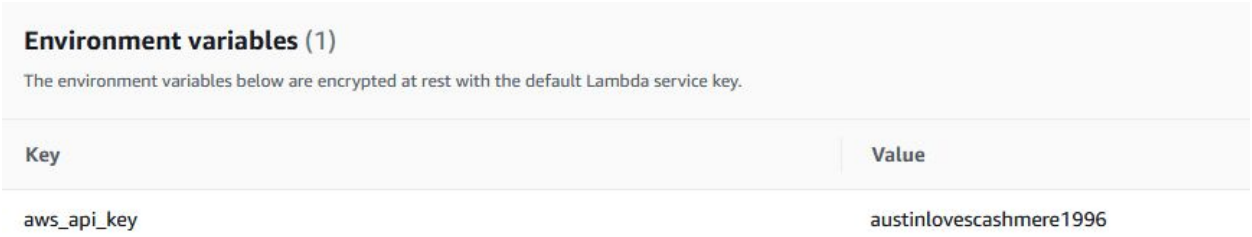
```
"Statement": [
     {
          "Effect": "Allow",
          "Principal": {
               "AWS": "*"
          },
          "Action": "lambda:InvokeFunction",
          "Resource":
"arn:aws:lambda:us-east-1:01234567892:function:HelloWorld"
     }
```

05. **Exposed credentials in environment variables**

By default AWS Lambda environment variables are encrypted at rest on the container

environment. However, the AWS console may reveal the plaintext credentials -- here too they should be encrypted.

Example:



AWS Elastic Load Balancer and Application Load Balancer

AWS Load balancers distribute traffic to listeners (usually EC2 instances). AWS has two primary load balancers: Elastic and Application. AWS ELB is designed for layer 4 and AWS ALB is designed for layer 7. Utilizing AWS load balancers is a good defense in depth measure and features high availability. ELB offers the flexibility to centrally manage TLS settings, also TLS termination, and offload CPU workloads. By default, AWS load balancers are good defense against layer 7 and layer 4 (D)Dos attacks. For example, thread-based web servers, such as Apache may be prone to 'slow' DOS attacks, such as slow loris; placing ALB in front of the Apache web server can mitigate slow loris attacks as ALB will not forward incomplete requests to the Apache webserver. In the case of layer 4 attacks, ELB can simply route the traffic to different webservers hosted on independent EC2 instances via listeners, dividing the load. Furthermore, ELB can be configured to accept only properly formed requests which will thwart classic SYN-based and UDP reflection attacks.

Load Balancer Security Checklist

01. Request Smuggling

ALB uses Keep-Alive to route multiple requests from different senders in a single TCP connection. By default, ALB and ELB can be vulnerable to request smuggling

depending on the backend system. AWS introduced a fix which can be implemented by enabling the "routing.http.drop_invalid_header_fields.enabled" flag; however, it's known to break a lot of services and needs to be explicitly enabled. Enabling the ALB request smuggling protection drops ambiguous headers for TE/CL. Services using AWS load balancers should be manually tested for request smuggling if the flag is disabled. If the customer is using Nginx, the 'ignore_invalid_headers' flag will thwart malicious headers from being sent to the backend. For more info, read here: API LoadBalancerAttribute

02. Encryption in transit

Verify negotiation configuration isn't using insecure and deprecated protocols/ciphers. Also verify all listeners are using secure protocols.

Example: Here is a vulnerable listener configuration for a load balancer, allowing HTTP.

```
{
            "Listener": {
                "InstancePort": 80,
                "LoadBalancerPort": 80,
                "Protocol": "HTTP",
                "InstanceProtocol": "HTTP"
            },
```

AWS Auto Scaling Groups

Autoscaling groups are logical groups of EC2 instances. Autoscaling groups are often created with AMIs (Amazon Machine Images, just a copy of an EC2 instance). Auto scaling groups are a great asset for availability as they automatically scale, monitor the health of instances, and will deploy a new instance upon failure.

ASG Security Checklist:

1. **Over-permissive IAM Policy**
   IAM roles can be assigned during launch configuration of ASGs, like independent EC2 instances, this should be checked.

AWS CloudFront

CloudFront is AWS's content discovery network.

CloudFront Security Checklist

1. **Encryption in transit**
   a. Verify the CloudFront distribution's viewer protocol policy only allows HTTPS. A viewer policy set to "allow-all" is flawed, as it allows HTTP.
   b. Verify the CloudFront distribution's origin protocol policy only allows HTTPS. An origin policy set to "OriginProtocolPolicy": "http-only" is flawed, as it allows HTTP.
   c. Verify negotiation configuration isn't using insecure and deprecated protocols/ciphers. Also verify all listeners are using secure protocols.