**1. Alarms with no Action Configured**

* **Severity:** Low
* **Description:** AWS CloudWatch Alarms have been created to monitor specific metrics (e.g., CPU utilisation, error rates, specific log patterns), but they are not configured to trigger any action (like sending a notification via SNS, triggering an Auto Scaling action, or invoking a Lambda function) when they enter the ALARM state.
* **Impact:** If a monitored threshold is breached or a critical event occurs, administrators or automated systems will not be notified. This can lead to delayed responses to performance issues, security events, application errors, or potential cost overruns, potentially increasing the duration and severity of an incident. The monitoring effort is wasted if no one is alerted.
* **Remediation Steps:**
  1. Identify the purpose of each configured CloudWatch Alarm.
  2. For alarms monitoring critical conditions, configure appropriate actions.
  3. Common actions include:
     + Sending a notification to an SNS (Simple Notification Service) topic subscribed to by administrators or ticketing systems.
     + Triggering an EC2 Auto Scaling action (e.g., scale out/in).
     + Initiating an EC2 action (e.g., stop, terminate, reboot).
     + Invoking a Lambda function for custom responses.
  4. Ensure the configured actions (like SNS topics) are functional and notifications reach the intended recipients.

**2. EBS Encryption Disabled by Default**

* **Severity:** Low
* **Description:** The AWS account setting to automatically encrypt all new EBS (Elastic Block Store) volumes created within a specific region is not enabled. This means new volumes will be unencrypted unless encryption is explicitly specified during creation.
* **Impact:** Data stored on unencrypted EBS volumes is not protected at rest by default. While AWS has strong physical security, regulatory or compliance requirements (like PCI-DSS, HIPAA) often mandate encryption at rest. In the unlikely event of **unauthorised** physical access to the storage media or certain types of underlying system compromises, unencrypted data could theoretically be exposed. It primarily represents a compliance and **defence**-in-depth gap.
* **Remediation Steps:**
  1. Navigate to the EC2 Dashboard in the AWS Management Console for each relevant region.
  2. Go to "EC2 Settings" (or similar, depending on console updates) and find the "EBS encryption" option.
  3. Enable "Encrypt new EBS volumes by default".
  4. Choose a default KMS key (AWS managed aws/ebs key or a customer-managed key).
  5. **Note:** This only affects *newly created* volumes. Existing unencrypted volumes must be encrypted manually (e.g., snapshot -> copy snapshot with encryption enabled -> create new volume from encrypted snapshot -> attach new volume).

**3. Unused Credentials for 90 Days not Disabled**

* **Severity:** Medium/High
* **Description:** IAM (Identity and Access Management) users exist within the account that possess active access keys or console passwords which have not been used for 90 days or more. There is no automated process in place to disable these inactive credentials.
* **Impact:** Stale or unused credentials represent a significant security risk. If these credentials (especially access keys potentially embedded in old code, configuration files, or leaked) are compromised, they provide a valid entry point for attackers into the AWS environment, even long after the user stopped needing them. The principle of least privilege includes **minimising** the *time* credentials are active. The impact is higher if the unused credentials belong to users with significant privileges.
* **Remediation Steps:**
  1. Regularly generate and review the IAM Credential Report from the IAM console.
  2. Identify users whose password\_last\_used or access\_key\_N\_last\_used\_date is older than 90 days (or your defined threshold).
  3. For users with unused console passwords, consider disabling console access or deleting the user if no longer needed.
  4. For users with unused access keys:
     + Make the unused access key inactive first.
     + After confirming no legitimate process breaks, delete the inactive key.
     + If both keys are unused and the user doesn't need programmatic access, delete both keys.
  5. Implement an automated process (e.g., using AWS Config rules, Lambda functions triggered by EventBridge, or custom scripts) to detect and potentially disable or notify about unused credentials.

**4. Cross-Account AssumeRole Policy Lacks MFA**

* **Severity:** (Not specified, typically Medium/High)
* **Description:** An IAM role exists whose trust policy allows principals (users or roles) from *another* AWS account to assume it (sts:AssumeRole), but the policy does not include a condition requiring that the assuming principal's session was authenticated using Multi-Factor Authentication (MFA).
* **Impact:** If an IAM user's credentials (password and/or access keys) in the *trusted* account are compromised, an attacker can assume this role in the *trusting* account *without* needing to bypass MFA. This effectively weakens the security posture, as MFA is a critical control for preventing unauthorised access, especially for privileged cross-account operations.
* **Remediation Steps:**
  1. Identify the IAM role(s) allowing cross-account access.
  2. Edit the role's Trust Relationship (AssumeRolePolicyDocument).
  3. Add or modify the Condition block within the sts:AssumeRole statement to include the requirement for MFA:

JSON

"Condition": {

"Bool": {

"aws:MultiFactorAuthPresent": "true"

}

}

* 1. Ensure that users in the trusted account who need to assume this role are configured with and use MFA when authenticating initially.

**5. Root Account without Hardware MFA**

* **Severity:** High
* **Description:** The AWS account's root user is not protected by a *hardware* Multi-Factor Authentication (MFA) device. It might have no MFA enabled at all, or potentially only a virtual MFA device (authenticator app). Best practice strongly recommends a hardware MFA for the root user.
* **Impact:** The root user has unrestricted access to all resources and settings within the AWS account, including billing information and the ability to close the account or change support plans. Compromise of the root account credentials (email/password) without the robust protection of hardware MFA could lead to a complete account takeover, catastrophic data loss, resource deletion, significant financial damage, and reputational harm. Hardware MFA is less susceptible to phishing and device compromise than virtual MFA.
* **Remediation Steps:**
  1. Immediately log in as the root user.
  2. Navigate to "My Security Credentials" from the account dropdown menu.
  3. Expand the "Multi-factor authentication (MFA)" section.
  4. Click "Activate MFA".
  5. Select "Hardware MFA device" (or U2F Security Key if preferred and compatible).
  6. Follow the instructions to register a purchased, compatible hardware MFA token (e.g., YubiKey, Gemalto).
  7. **Crucially:** Store the hardware MFA device securely and separately from root account credentials. Record the device serial number.
  8. **Minimise** the use of the root account; use IAM roles/users with appropriate permissions for all routine administrative tasks.

**6. Users with Passwords and Keys Enabled**

* **Severity:** Medium
* **Description:** IAM users within the account have been configured with *both* a console login password *and* active programmatic access keys (Access Key ID and Secret Access Key).
* **Impact:** This configuration increases the attack surface for that user identity. If either the password *or* the access key is compromised, an attacker gains access corresponding to the user's permissions. Often, users only require one method of access (e.g., administrators primarily use the console, application service accounts only need keys). Enabling both when only one is needed violates the principle of least privilege regarding credential types and can indicate poor credential management hygiene.
* **Remediation Steps:**
  1. Review IAM users who have both password and access keys enabled.
  2. Determine the actual access requirements for each user.
  3. If a user only needs console access (human user):
     + Make their access key(s) inactive.
     + After a validation period, delete the inactive key(s).
  4. If a user only needs programmatic access (application/service user):
     + Disable their console login password.
  5. Ensure users requiring console access also have MFA enabled.

**7. Bucket Access Logging Disabled**

* **Severity:** Low
* **Description:** An Amazon S3 bucket does not have Server Access Logging enabled. These logs provide detailed records for requests made against the bucket.
* **Impact:** Without access logs, it is difficult or impossible to perform security audits, investigate potential **unauthorised** access attempts, troubleshoot access issues, or understand usage patterns for the bucket. This lack of visibility hinders incident response and compliance efforts.
* **Remediation Steps:**
  1. Navigate to the S3 service in the AWS Management Console.
  2. Select the relevant bucket and go to its "Properties" tab.
  3. Find the "Server access logging" setting and click "Edit".
  4. Enable logging.
  5. Specify a target S3 bucket where the logs should be delivered (cannot be the same bucket). Ensure the target bucket exists and appropriate permissions are granted to the S3 log delivery service principal.
  6. Optionally specify a prefix for log file **organisation** within the target bucket.
  7. Consider setting up lifecycle policies on the target log bucket to manage log retention and storage costs.

**8. Bucket Allowing HTTP Communication**

* **Severity:** Low
* **Description:** The S3 bucket's configuration (e.g., bucket policy) does not enforce encrypted connections (HTTPS). It allows clients to access the bucket over unencrypted HTTP.
* **Impact:** Data transferred to or from the S3 bucket via HTTP is sent in plaintext over the network. This makes the data vulnerable to eavesdropping or Man-in-the-Middle (MitM) attacks if intercepted on an insecure network segment between the client and AWS. This is particularly risky if sensitive data is stored in the bucket.
* **Remediation Steps:**
  1. Navigate to the S3 service and select the relevant bucket.
  2. Go to the "Permissions" tab and edit the "Bucket policy".
  3. Add a statement to the policy that explicitly denies requests that are not using HTTPS. This is done using the aws:SecureTransport condition key:

JSON

{

"Sid": "EnforceSSL",

"Effect": "Deny",

"Principal": "\*",

"Action": "s3:\*",

"Resource": [

"arn:aws:s3:::YOUR-BUCKET-NAME/\*",

"arn:aws:s3:::YOUR-BUCKET-NAME"

],

"Condition": {

"Bool": {

"aws:SecureTransport": "false"

}

}

}

*(Ensure this statement is correctly integrated into your existing policy, if any)*

* 1. Test access to ensure legitimate clients using HTTPS still work, while HTTP attempts are blocked.

**9. Bucket without MFA Delete**

* **Severity:** Low
* **Description:** The MFA Delete feature is not enabled for the S3 bucket. MFA Delete requires users to provide a valid MFA code when attempting to permanently delete an object version or change the versioning state of the bucket.
* **Impact:** Without MFA Delete, accidental or malicious deletion of object versions (if versioning is enabled) or changes to the bucket's versioning status can be performed by anyone possessing credentials with the necessary permissions (even the root user) without an additional MFA challenge. This increases the risk of irreversible data loss.
* **Remediation Steps:**
  1. **Prerequisite:** Ensure bucket versioning is enabled on the target bucket. MFA Delete requires versioning.
  2. MFA Delete must be enabled by the **root account user** using the AWS CLI or SDKs (it cannot be enabled via the console).
  3. The root user needs an MFA device configured.
  4. Use the put-bucket-versioning CLI command:

Bash

aws s3api put-bucket-versioning --bucket YOUR-BUCKET-NAME \

--versioning-configuration Status=Enabled,MFADelete=Enabled \

--mfa "arn:aws:iam::ACCOUNT-ID:mfa/ROOT-USERNAME MFA-CODE"

Replace YOUR-BUCKET-NAME, ACCOUNT-ID, ROOT-USERNAME, and provide the current MFA-CODE from the root user's MFA device.

**10. Bucket without Versioning**

* **Severity:** Low
* **Description:** S3 bucket versioning is not enabled for the bucket. Versioning preserves, retrieves, and restores every version of every object stored in the bucket.
* **Impact:** Without versioning, if an object is deleted or overwritten (e.g., by accident, application error, or malicious action), the previous data is permanently lost. Versioning provides a critical safety net against data loss and facilitates easier recovery.
* **Remediation Steps:**
  1. Navigate to the S3 service and select the relevant bucket.
  2. Go to the "Properties" tab.
  3. Find the "Bucket Versioning" setting and click "Edit".
  4. Select "Enable" and save changes.
  5. **Note:** Enabling versioning can increase storage costs as multiple object versions are retained. Implement S3 Lifecycle policies to manage noncurrent versions (e.g., transition to cheaper storage tiers or expire after a certain period) to control costs.

**11. DKIM not enabled**

* **Severity:** Low
* **Description:** DomainKeys Identified Mail (DKIM) signing is not configured in AWS Simple Email Service (SES) for the domain(s) used to send email. DKIM adds a digital signature to outgoing emails, allowing receivers to verify the email originated from an **authorised** source.
* **Impact:** Emails sent without DKIM signatures are more likely to be flagged as spam or rejected by receiving email servers, negatively impacting email deliverability. It also slightly increases the risk of successful email spoofing using the sender's domain.
* **Remediation Steps:**
  1. Navigate to the SES console.
  2. Go to "Verified identities" (or similar section for domains).
  3. Select the domain identity.
  4. Go to the "Authentication" tab.
  5. In the DKIM section, click "Generate DKIM settings" (or similar action).
  6. SES will provide CNAME records. Add these records to your domain's DNS configuration via your DNS provider.
  7. Wait for DNS propagation, then SES should automatically detect the records and show DKIM status as "Verified".

**12. DKIM not verified**

* **Severity:** Low
* **Description:** DKIM settings have been generated in SES for a domain, but the corresponding DNS CNAME records have not been correctly published in the domain's DNS zone, or they haven't propagated fully. SES cannot verify the DKIM configuration.
* **Impact:** Same as "DKIM not enabled". DKIM signing is not active for outgoing emails, leading to potential deliverability issues and increased spam classification risk.
* **Remediation Steps:**
  1. In the SES console, under the domain's "Authentication" tab, find the required DKIM CNAME records.
  2. Verify that these records have been accurately added to your domain's DNS settings with your DNS provider (check for typos, correct record type CNAME, correct hostnames and values).
  3. Use DNS lookup tools (e.g., dig, nslookup, online tools) to confirm the CNAME records are publicly resolvable.
  4. Allow sufficient time for DNS propagation (can take up to 72 hours, but usually much faster).
  5. Once DNS records are confirmed to be correct and propagated, the DKIM status in the SES console should eventually update to "Verified". You might need to click a "Retry" or "Verify" button if available.

**13. CE-Ansible-Queue SendMessage Authorised to All Principles**

* **Severity:** High
* **Description:** An SQS (Simple Queue Service) queue, presumably named CE-Ansible-Queue, has a resource-based policy granting sqs:SendMessage permission to all principals ("Principal": "\*" or "Principal": {"AWS": "\*"}).
* **Impact:** This configuration allows *any* AWS identity (any user, role, or service within any AWS account, and potentially even unauthenticated users depending on other factors) to send messages to this queue. This can lead to:
  + Injection of malicious or unintended data/commands processed by the queue consumer (Ansible).
  + Denial of Service (DoS) by flooding the queue.
  + Increased SQS costs.
  + Compromise of downstream systems if the consumer blindly trusts messages.
* **Remediation Steps:**
  + Navigate to the SQS service in the AWS console.
  + Select the CE-Ansible-Queue.
  + Go to the "Access policy" tab and click "Edit".
  + Modify the policy statement(s) that grant sqs:SendMessage.
  + Change the "Principal" element from a wildcard ("\*") to specific, known AWS Principal ARNs (e.g., arn:aws:iam::ACCOUNT-ID:role/SpecificRoleThatNeedsToSend) that legitimately need to send messages to this queue.
  + Apply the principle of least privilege – only grant access to entities that absolutely require it.
  + Consider adding Condition clauses (e.g., aws:SourceAccount or aws:SourceArn) for further restriction if applicable.

**14. Egress Traffic Allow to Some Ports**

* **Severity:** Low (but context-dependent)
* **Description:** Security Groups or Network ACLs associated with resources (like EC2 instances) permit outbound (egress) traffic to the internet (e.g., 0.0.0.0/0) or other networks on specific ports that might not be strictly necessary for the resources' function. The specific ports were likely identified in the report (e.g., 22, 25, 135, 445, 3389).
* **Impact:** Overly permissive egress rules increase the attack surface. If a resource is compromised (e.g., via malware or an exploit), these open egress ports can be used by the attacker for:
  + Data exfiltration to external servers.
  + Connecting back to Command and Control (C2) servers.
  + Scanning or attacking other systems on the internet or internal networks. Restricting egress traffic to only known-good destinations and ports limits the potential damage of a compromise (**defence**-in-depth).
* **Remediation Steps:**
  + Identify the Security Groups (primarily) and Network ACLs associated with the affected resources.
  + Review the Egress (Outbound) rules.
  + For each rule allowing traffic (especially to 0.0.0.0/0), determine if it is absolutely necessary for the application/instance function (e.g., outbound HTTPS/443 for API calls or OS updates is common; outbound SSH/22 to anywhere is usually risky).
  + Modify the rules to be more specific:
    - Restrict destination IP addresses/ranges instead of using 0.0.0.0/0 where possible (e.g., specific update servers, partner APIs).
    - Allow only the specific ports required.
  + Consider implementing a default deny for all egress traffic and only explicitly allowing necessary flows.
  + Investigate using VPC Endpoints (Interface or Gateway) for accessing AWS services privately without needing egress rules to the public internet.
  + **Utilise** tools like VPC Flow Logs to **analyse** actual egress traffic patterns to inform rule tightening.