AWS Attack & Defense

How do you secure an AWS EC2 instance?

I secure EC2 instances through multiple layers.

Network security:

- Place instances in private subnets with no direct internet access.
- Use security groups allowing only necessary inbound traffic from specific sources (never 0.0.0.0/0 on SSH/RDP).
- Implement NACLs as additional subnet-level protection.
- Use VPC endpoints for AWS service access avoiding public internet.

Access control:

- Disable password authentication for SSH using key pairs only.
- Implement Session Manager for administrative access eliminating SSH key management and providing audit trails.
- Use IAM instance profiles for application credentials rather than hardcoded keys.
- Implement MFA for any privileged access.

Hardening:

- Use minimal AMIs reducing attack surface.
- Disable unnecessary services and ports.
- Implement host-based firewalls.
- Apply CIS benchmarks for OS hardening.
- Keep instances patched using Systems Manager Patch Manager.

Encryption:

- Enable EBS encryption for all volumes including root.
- Encrypt data in transit with TLS.
- Use KMS for key management.

Monitoring:

- Enable detailed CloudWatch monitoring.
- Install CloudWatch agent for custom metrics and logs.
- Enable VPC Flow Logs.
- Use GuardDuty for threat detection.

- Implement Inspector for vulnerability scanning.

Configuration management:

- Use immutable infrastructure where instances aren't patched but replaced.
- Implement configuration as code.
- Use golden AMIs with security controls baked in.

Backup:

- Enable automated EBS snapshots with encryption.
- Test recovery procedures.
- Implement disaster recovery plans.

The principle is defense in depth with multiple security layers.

What is AWS Identity and Access Management (IAM), and how does it work?

IAM is AWS's authentication and authorization service controlling who can access which AWS resources and what actions they can perform. It works through several components.

- **Users** represent individuals or applications with permanent credentials.
- **Groups** are collections of users with common permissions.
- **Roles** are assumed by users or services providing temporary credentials—these are preferred over users for applications.
- **Policies** are JSON documents defining permissions specifying allowed or denied actions on specific resources. Policies can be identity-based (attached to users/groups/roles) or resource-based (attached to resources like S3 buckets).
- **Authentication** verifies identity through credentials—passwords, access keys, or federated identity from external identity providers via SAML or OIDC.
- **Authorization** evaluates policies when requests are made—IAM checks all applicable policies (identity-based, resource-based, SCPs, permission boundaries) using explicit deny-first logic where any deny overrides allows.
- **Temporary credentials** through STS provide time-limited access via role assumption, improving security over long-lived access keys.
- **MFA** adds additional authentication factor.
- **Condition elements** in policies enforce additional constraints like IP ranges, time windows, or MFA requirements.

IAM is global and free, providing centralized access control across all AWS services. Best practices include principle of least privilege, using roles over users, enabling MFA, rotating credentials, and regular access reviews. IAM is foundational to AWS security—getting it wrong exposes your entire environment.

How can you protect against DDoS attacks in AWS?

AWS provides multiple layers of DDoS protection.

- **AWS Shield Standard** is automatic, free protection included with all AWS services, defending against common network and transport layer attacks (SYN floods, UDP reflection attacks). It uses advanced traffic engineering and proprietary mitigation techniques protecting AWS infrastructure.
- **AWS Shield Advanced** is a paid service providing enhanced protection, 24/7 access to AWS DDoS Response Team (DRT), cost protection from scaling during attacks, and advanced real-time metrics and reporting.

For **application architecture**, I implement:

- Auto-scaling to absorb attack traffic.
- CloudFront CDN distributing traffic across edge locations making volumetric attacks less effective.
- Route 53's anycast network for DNS DDoS resilience.
- Deployment behind Application Load Balancers or Network Load Balancers which provide inherent DDoS protection.

AWS WAF protects against application-layer attacks by:

- Filtering malicious HTTP requests.
- Implementing rate limiting to prevent overwhelming applications.
- Blocking requests from known malicious IPs.
- Using managed rule groups for common attack patterns.

Operational practices include:

- Monitoring with CloudWatch for traffic anomalies.
- Setting up CloudWatch alarms for unusual metrics.
- Having incident response procedures for DDoS events.
- Regularly testing with controlled load testing.

Architecture patterns:

- Avoid single points of failure.

- Implement geographic distribution.
- Use multiple availability zones.
- Design for elastic scalability.

For critical applications, I implement Shield Advanced with DRT engagement plan, WAF with strict rate limiting and geographic restrictions, and CloudFront with origin protection preventing direct origin access. The combination makes applications resilient to most DDoS attacks.

What is AWS GuardDuty, and how does it help in security?

GuardDuty is a threat detection service that continuously monitors AWS accounts for malicious activity and unauthorized behavior. It analyzes multiple data sources:

- **CloudTrail event logs** detecting unusual API calls, failed authentication attempts, or suspicious changes to resources.
- **VPC Flow Logs** identifying unusual network traffic patterns, communication with known malicious IPs, or data exfiltration attempts.
- **DNS logs** detecting DNS queries to command-and-control servers or malicious domains.
- **EKS audit logs and runtime monitoring** for Kubernetes security.

GuardDuty uses **threat intelligence** from AWS Security, CrowdStrike, and Proofpoint, plus **machine learning** that establishes baselines of normal behavior and detects anomalies.

It generates **findings** when threats are detected, categorized by severity (low, medium, high) and type (compromised instance, reconnaissance, data exfiltration, etc.). Each finding includes detailed context—affected resources, threat indicators, and recommended remediation.

Benefits include:

- No infrastructure to manage (fully managed service).
- Continuous monitoring without agents.
- Intelligent detection reducing false positives.
- Integration with Security Hub and EventBridge for automated response.
- Multi-account support through Organizations integration.

To **use effectively**, I:

- Enable GuardDuty across all accounts and regions.
- Configure trusted IP lists and threat lists for customization.
- Integrate findings with SIEM for correlation.
- Implement automated response through EventBridge triggering Lambda functions for common

findings.

- Establish escalation procedures for high-severity findings.
- Regularly review findings tuning for false positives.

GuardDuty provides essential threat detection that would be extremely difficult to implement manually.

Explain the purpose of AWS CloudTrail and CloudWatch in security monitoring.

CloudTrail and **CloudWatch** serve complementary but distinct security monitoring purposes.

CloudTrail is the audit log service recording all API calls made in your AWS account—who made the call, when, from what IP address, what parameters were used, and what the response was. It provides:

- **Comprehensive audit trail** for compliance and forensics.
- **Detective control** enabling investigation of security incidents.
- **Governance** tracking changes to resources.
- **Compliance evidence** for audits.

CloudTrail logs should be enabled in all regions, sent to centralized S3 bucket with encryption and MFA Delete, have log file validation enabled ensuring integrity, and be integrated with CloudWatch Logs for real-time analysis.

CloudWatch is the monitoring and observability service with multiple security-relevant components:

- **CloudWatch Logs** aggregates application and system logs enabling searching and analysis.
- **Metric Filters** extract metrics from logs triggering alarms on security events like failed SSH attempts or unauthorized API calls.
- **CloudWatch Alarms** trigger on metric thresholds sending notifications or invoking automated responses.
- **CloudWatch Events/EventBridge** enables event-driven security automation.

For **security monitoring**, I use:

- CloudTrail to track who did what creating accountability.
- CloudWatch Logs for centralized log aggregation.
- Metric Filters to detect security events in logs.
- Alarms to notify on suspicious activity.

Together they provide **comprehensive visibility**: CloudTrail answers "who did what to which resource and when" while CloudWatch answers "what's the current state and are there anomalies." Integration enables real-time detection—CloudTrail logs stream to CloudWatch Logs where filters detect patterns triggering alarms that invoke Lambda for automated response.

What is AWS Key Management Service (KMS), and how does it handle encryption keys?

AWS KMS is a managed service for creating and controlling encryption keys used to encrypt data across AWS services and applications. KMS uses **envelope encryption** where data is encrypted with a data encryption key (DEK), and the DEK itself is encrypted with a KMS key (formerly called CMK).

KMS keys are the primary resources—they never leave KMS and all cryptographic operations happen within KMS's FIPS 140-2 validated hardware security modules (HSMs). KMS supports:

- **Symmetric keys** (same key for encryption and decryption, most common).
- **Asymmetric keys** (public/private key pairs for signing or encryption).

Key types include:

- AWS managed keys (created automatically by services, free).
- Customer managed keys (you create and control, full flexibility).
- AWS owned keys (used by services, invisible to you).

Key policies define who can use and manage keys, with additional IAM policies and grants providing granular access control. KMS provides **automatic key rotation** for customer managed symmetric keys, rotating annually while keeping old key material for decryption.

CloudTrail integration logs all key usage for audit trails. **Multi-region keys** enable encryption across regions with the same key material.

For **security**, KMS keys never leave KMS unencrypted, all operations are logged, access requires both key policy and IAM permissions, and deletion has mandatory waiting period preventing accidental key loss.

I use KMS for encrypting EBS volumes, S3 buckets, RDS databases, and application secrets, with separate keys per environment/application following least privilege. Key policies restrict usage to specific services and roles, and I enable automatic rotation. KMS is foundational for encryption at rest in AWS.

How do they differ?

I covered this in question 6, but I'll expand with AWS-specific details.

Security Groups are stateful, virtual firewalls at the instance (ENI) level. They use allow-list only—you specify what's permitted; everything else is denied. Being stateful means return traffic is automatically allowed regardless of outbound rules. Security groups evaluate all rules before deciding to allow traffic (no rule ordering), support referencing other security groups as sources (enabling micro-segmentation without IP management), and changes take effect immediately. You can have up to 5 security groups per instance with rules combined.

Network ACLs are stateless firewalls at the subnet level. They evaluate rules in numerical order stopping at first match, support both allow and deny rules, require explicit rules for both request and response traffic due to statelessness, and apply to all instances in the subnet. NACLs have separate inbound and outbound rule sets.

In practice, security groups are the primary traffic control—I create security groups per application tier (web, app, database) with specific rules. For example:

- Web security group allows 443 from 0.0.0.0/0.
- App security group allows 8080 from web security group.
- Database security group allows 3306 from app security group.

NACLs provide additional protection—blocking known malicious IPs, implementing deny rules for compliance, or adding subnet-level restrictions.

Key difference is security groups are more flexible and easier to manage (stateful, can reference other groups), while NACLs provide defense in depth and deny capabilities security groups lack. Both are evaluated for inbound traffic—NACL first, then security group.

How do you implement security best practices for AWS Lambda functions?

Lambda security requires attention to multiple areas.

IAM permissions:

- Create function-specific execution roles with only permissions needed.
- Avoid wildcard permissions.
- Use resource-based policies to control what can invoke the function.
- Implement least privilege rigorously since Lambda's serverless nature makes over-permissioning common.

Code security:

- Never hardcode secrets—use Secrets Manager or Parameter Store retrieving secrets at runtime.
- Implement input validation preventing injection attacks.
- Use dependency scanning checking for vulnerable libraries.
- Implement code signing ensuring only approved code executes.

Network isolation:

- Deploy Lambda in VPC when accessing VPC resources.
- Use private subnets with VPC endpoints for AWS service access avoiding NAT gateways.
- Implement security groups controlling Lambda's outbound connections.
- Use PrivateLink for third-party service access.

Environment variables:

- Encrypt sensitive values using KMS.
- Never store secrets in plaintext variables.
- Rotate secrets regularly.

Logging and monitoring:

- Enable CloudWatch Logs for all function executions.
- Implement structured logging with correlation IDs.
- Use CloudTrail to track function configuration changes.
- Set up alerts on errors or unusual invocation patterns.
- Use X-Ray for distributed tracing.

Runtime security:

- Keep runtimes updated to latest versions.
- Minimize function package size reducing attack surface.
- Implement short timeout values preventing runaway executions.
- Set appropriate memory limits.
- Use layers for common dependencies enabling centralized updates.

Triggers:

- Validate event sources.
- Implement authentication for HTTP triggers via API Gateway with IAM or Cognito.
- Use resource policies restricting what can invoke functions.
- Validate event data before processing.

The serverless model requires different security thinking—focus on IAM boundaries, temporary execution context, and event-driven attack vectors.

What is the AWS Well-Architected Framework, and why is it important for security?

The AWS Well-Architected Framework is a set of best practices across six pillars: Operational Excellence, Security, Reliability, Performance Efficiency, Cost Optimization, and Sustainability. The **Security Pillar** is specifically important for cloud security, covering identity and access management, detective controls, infrastructure protection, data protection, and incident response.

It's important because it provides:

- **Structured approach** to evaluating architectures against proven best practices.
- **Common language** for discussing security with stakeholders.
- **Comprehensive coverage** of security domains often overlooked.
- **Continuous improvement** through regular reviews.

The framework includes **design principles** like:

- Implementing security at all layers.
- Enabling traceability.
- Automating security best practices.
- Protecting data in transit and at rest.
- Keeping people away from data.
- Preparing for security events.

Best practices are detailed for each area with implementation guidance. **Well-Architected Tool** in AWS Console lets you conduct reviews, answer questions about your workload, and receive recommendations for improvement with links to documentation.

For **security specifically**, I use the framework during architecture design ensuring security is built-in, conduct Well-Architected Reviews quarterly identifying gaps, prioritize remediation based on framework recommendations, and use it for cross-team education establishing shared security understanding.

The framework prevents ad-hoc security decisions, provides comprehensive security checklist, and helps justify security investments with best-practice backing. Following Well-Architected principles significantly improves security posture and reduces risk of common security mistakes.

How do you securely manage secrets and credentials in AWS?

I never hardcode secrets or commit them to version control.

AWS Secrets Manager is the primary tool for managing database credentials, API keys, and other secrets. It provides:

- Automatic rotation for RDS, Redshift, and DocumentDB credentials.
- Encryption at rest with KMS.
- Fine-grained IAM permissions controlling access.
- Versioning enabling rollback.
- Integration with many AWS services.

I use Secrets Manager for credentials requiring rotation and high-value secrets.

Systems Manager Parameter Store is lighter-weight for configuration data and secrets, offering:

- Secure string parameters encrypted with KMS.
- Hierarchical storage organizing parameters.
- Versioning.
- Lower cost (standard parameters are free).

I use Parameter Store for application configuration and lower-sensitivity secrets.

IAM roles eliminate long-lived credentials entirely—applications running on EC2, Lambda, or ECS assume roles receiving temporary credentials automatically rotated.

For implementation:

- Applications retrieve secrets at runtime via SDK calls rather than environment variables.
- I implement caching to reduce API calls and costs while respecting rotation periods.
- Use resource-based policies restricting which resources can access specific secrets.
- Enable CloudTrail logging of secret access for audit trails.
- Implement least privilege where each application can only access its own secrets.
- Use secret rotation for database credentials and API keys.

Environment variables encrypted with KMS are acceptable for less sensitive configuration but never for high-value secrets.

For CI/CD, I use OIDC federation with GitHub Actions or similar providing short-lived credentials rather than storing AWS access keys in CI systems.

Key rotation happens automatically for Secrets Manager-managed secrets, and I implement custom Lambda functions for rotating third-party API keys.

This approach eliminates static credentials, provides audit trails, and enables centralized secret lifecycle management.

Enforce TLS 1.2+ on all external applications in a cloud environment, and why is this important for security?

Enforcing TLS encryption for all external communications is critical to prevent eavesdropping, man-in-the-middle attacks, and data tampering.

Implementation:

For applications behind Application Load Balancers:

- Configure HTTPS listeners with certificates from AWS Certificate Manager (ACM).
- Select security policies requiring TLS 1.2 minimum (ELBSecurityPolicy-TLS-1-2-2017-01 or newer).
- Configure HTTP listeners to redirect to HTTPS (301 or 302 redirects).
- Disable weaker protocols.

For CloudFront distributions:

- Select TLSv1.2_2021 or newer security policy.
- Configure custom SSL certificates via ACM.
- Set "Viewer Protocol Policy" to "Redirect HTTP to HTTPS" or "HTTPS Only".
- Configure minimum SSL/TLS version.

For API Gateway:

- Select TLS 1.2 minimum in domain configuration.
- Enforce HTTPS through resource policies.

S3 buckets require encryption in transit via bucket policies with `aws:SecureTransport` condition denying requests over HTTP.

For **direct EC2 applications**:

- Configure web servers (nginx, Apache) with modern TLS configurations.
- Obtain certificates from ACM or Let’s Encrypt.
- Disable SSLv3 and TLS 1.0/1.1.

- Use strong cipher suites preferring AEAD ciphers.

Validation includes automated scanning with tools like SSL Labs, Config rules checking for TLS enforcement, and penetration testing verifying only modern protocols work.

Why it’s critical:

- TLS 1.2+ provides strong encryption protecting data in transit.
- Prevents passive eavesdropping on sensitive data.
- Mitigates man-in-the-middle attacks.
- Provides server authentication preventing impersonation.
- Meets compliance requirements (PCI DSS mandates TLS 1.2+).

Older protocols like TLS 1.0/1.1 have known vulnerabilities and should be disabled. Modern TLS with forward secrecy protects historical traffic even if keys are later compromised.

How can you protect against data exfiltration in a cloud environment?

Data exfiltration protection requires multiple defensive layers.

Network controls:

- Implement VPC endpoints for AWS services keeping traffic on AWS private network.
- Use PrivateLink for third-party services.
- Restrict outbound internet access through NAT gateways with limited security groups.
- Implement DNS filtering blocking known malicious domains.
- Use VPC Flow Logs to monitor unusual outbound connections.

DLP (Data Loss Prevention):

- Use Amazon Macie to discover and classify sensitive data in S3.
- Implement bucket policies preventing unauthorized access.
- Enable MFA Delete on critical buckets.
- Scan data movement for PII or sensitive patterns.

IAM restrictions:

- Implement least privilege limiting who can access sensitive data.
- Use permission boundaries preventing privilege escalation.
- Require MFA for sensitive operations.
- Use SCPs to prevent disabling of logging or creating external access.

Monitoring and detection:

- Use GuardDuty which detects unusual data transfer patterns and communication with known command-and-control servers.
- Implement CloudWatch alarms on anomalous data transfer volumes.
- Monitor CloudTrail for suspicious data access patterns (unusual API calls, access from new locations).
- Use VPC Flow Logs to detect large outbound data transfers.

Data protection:

- Encrypt data at rest making exfiltrated data useless without keys.
- Implement bucket versioning and Object Lock preventing data destruction.
- Use S3 Access Points limiting how data can be accessed.
- Implement cross-region replication with separate accounts for backup integrity.

Detective controls:

- Establish baselines of normal data access patterns.
- Alert on deviations like bulk downloads or access from unusual locations/times.
- Integrate with SIEM for correlation.

Incident response: have playbooks for suspected exfiltration including immediate credential rotation, blocking suspicious network connections, and forensic evidence collection.

Prevention is difficult because authorized users legitimately access data—focus on detection and rapid response.

What is a privilege escalation attack, and how do you prevent it in a cloud environment?

Privilege escalation is when an attacker with limited permissions gains higher privileges they weren't intended to have. In AWS, this might involve a user with `iam:PutUserPolicy` permission granting themselves administrator access, or someone with `iam:PassRole` creating resources with more privileged roles.

Common escalation paths include:

- IAM permissions allowing policy modification (`iam:PutUserPolicy`, `iam:AttachUserPolicy`).
- Role assumption with overly broad trust policies.
- PassRole permission with unrestricted role passing.

- Lambda/EC2 creation permissions allowing highly privileged roles to be assigned.
- Updating existing resources (Lambda functions, EC2 user data) to execute malicious code with their existing privileges.

Prevention strategies:

- Implement **permission boundaries** limiting maximum permissions users can grant preventing escalation beyond boundaries.
- Use **SCPs** enforcing organizational guardrails that can't be bypassed.
- Apply **least privilege** rigorously so users only have minimum needed permissions.
- Implement **separation of duties** where no single user can complete sensitive workflows alone.
- Use **IAM Access Analyzer** to detect overly permissive policies and external access.

Specific controls:

- Restrict IAM modification permissions heavily.
- Require multiple approvals for IAM changes.
- Implement condition statements limiting when/how dangerous permissions can be used (MFA requirements, IP restrictions).
- Use **resource tags** with condition statements preventing manipulation of high-privilege resources.
- Avoid wildcard resources in policies.

Detection:

- Monitor CloudTrail for suspicious IAM changes.
- Alert on new permissions being granted especially to self.
- Detect creation of new access keys or roles.
- Use GuardDuty which has specific detections for privilege escalation attempts.

Regular audits:

- Review IAM permissions for escalation paths.
- Test with tools like Cloudsplaining or PMapper that identify risky permission combinations.
- Conduct purple team exercises attempting escalation.

Privilege escalation is one of the most common cloud attack techniques requiring proactive prevention.

Explain the importance of web application firewalls (WAFs) in cloud security.

WAF provides application-layer (Layer 7) protection that network firewalls and security groups can't provide. **AWS WAF** inspects HTTP/HTTPS requests protecting against OWASP Top 10 vulnerabilities—SQL injection, cross-site scripting (XSS), directory traversal, and others. It sits in front of CloudFront, ALB, API Gateway, or AppSync analyzing requests before they reach applications.

Core capabilities include:

- **Managed rule groups** from AWS and third-party providers (Fortinet, F5) covering common attack patterns and updated automatically.
- **Custom rules** for application-specific logic like rate limiting per IP or geo-blocking.
- **Rate-based rules** preventing DDoS and brute force attacks.
- **IP reputation lists** blocking known malicious sources.
- **Bot control** identifying and managing automated traffic.

Why it's important:

- Applications have vulnerabilities that can't be completely eliminated—WAF provides defense-in-depth, blocking exploit attempts even against unknown application vulnerabilities.
- It prevents **zero-day exploitation** through generic protections against entire attack classes.
- WAF enables **virtual patching** where known vulnerabilities in applications can be mitigated via WAF rules while permanent fixes are developed, critical during vulnerability disclosure windows.
- It provides **compliance support** for PCI DSS and other frameworks requiring WAF.

Implementation:

- Deploy WAF on all internet-facing applications.
- Start with AWS Managed Rules core rule set plus relevant specialty rules (SQL database, Linux, WordPress depending on stack).
- Implement custom rules for application-specific attacks or business logic abuse.
- Enable logging to S3 or Kinesis for analysis.
- Use count mode initially to tune rules reducing false positives, then switch to block mode.
- Implement rate limiting to prevent abuse.
- Use sampled requests for ongoing tuning.

Limitations: WAF can't protect against all attacks (business logic flaws, authentication bypasses), generates false positives requiring tuning, and adds slight latency. Despite limitations, WAF is essential defense layer for web applications.

How can you detect and respond to insider threats in a cloud environment?

Insider threats are challenging because insiders have legitimate access. Detection requires understanding normal behavior and identifying anomalies.

Behavioral analysis:

- Establish baselines of normal access patterns—what data each user typically accesses, from where, at what times.
- Use machine learning or manual review to detect deviations like bulk data downloads, access to new sensitive resources, or activity at unusual hours.

Access monitoring:

- Enable comprehensive CloudTrail logging across all accounts.
- Use GuardDuty which detects anomalous API activity.
- Monitor for privilege escalation attempts or IAM changes.
- Track data access patterns in S3 with server access logs and CloudTrail data events.
- Implement Macie to detect unusual data access or movement.

Specific indicators:

- Unusual geographic locations or IP addresses.
- Access spikes or bulk operations.
- Attempts to disable logging or security controls.
- Creation of unauthorized access methods (new IAM users, access keys).
- Copying data to external accounts or personal storage.
- Accessing resources outside normal job duties.

Prevention through controls:

- Implement least privilege limiting blast radius.
- Require MFA for sensitive operations.
- Use break-glass procedures for emergency access with comprehensive logging.
- Implement separation of duties preventing single-user complete workflows.
- Use data classification with stricter controls on sensitive data.

Technical controls:

- Enable MFA delete on critical S3 buckets.
- Implement SCPs preventing certain dangerous actions.

- Use permission boundaries.
- Enable encryption with separate key management.
- Implement VPC endpoints preventing data exfiltration to personal accounts.

Response procedures:

- When suspected insider threat detected, preserve evidence immediately through snapshots and log exports.
- Do not alert suspected insider to avoid evidence destruction.
- Engage legal and HR following established procedures.
- Rotate credentials they had access to.
- Conduct thorough investigation reviewing all their historical access.

Cultural aspects: positive security culture, clear acceptable use policies, regular security awareness training, and off-boarding procedures immediately revoking access.

Insider threats require balancing trust with verification.

How would you identify and rectify such misconfigurations?

I covered a similar scenario in question 12, but here's another specific AWS example.

A DevOps team was granted `iam:PassRole` with `Resource: to allow creating EC2 instances with instance profiles. However, they could pass *any` role, including a highly privileged administrator role created for a different purpose. A developer unknowingly launched an EC2 instance with the admin role for convenience during troubleshooting. Their instance was later compromised through an application vulnerability, and the attacker had full AWS account access through the admin instance profile.

Identification:

- IAM Access Analyzer would flag the overly broad PassRole permission.
- Prowler or similar tools scanning for wildcard resources would detect it.
- Manual policy review during security audits would catch it.
- Post-incident, CloudTrail logs showed the EC2 instance using the admin role making unusual API calls.
- GuardDuty detected anomalous behavior from the compromised instance.

Rectification:

- Immediately rotate credentials and terminate the compromised instance.

- Implement strict PassRole policy allowing only passage of specific approved roles with condition statement: `"Condition": {"StringEquals": {"iam:PassedToService": "ec2.amazonaws.com"}, "StringLike": {"iam:AssociatedResourceARN": "arn:aws:iam::ACCOUNT:role/DevOpsEC2Role"}}`.
- Create role naming conventions and organizational policy requiring specific prefixes for different purposes.
- Implement permission boundaries on roles that can be passed limiting their maximum permissions.
- Require peer review for all IAM policy changes.
- Use SCPs preventing attachment of admin policies to instance profiles.
- Conduct regular IAM policy audits with automated tooling.

Lessons: PassRole is particularly dangerous requiring strict control, wildcard resources in IAM policies should be rare exceptions requiring security team approval, and defense in depth means even compromised instances shouldn't have admin access.

What is a Distributed Denial-of-Service (DDoS) attack, and how can cloud providers help mitigate it?

DDoS attacks attempt to make services unavailable by overwhelming them with traffic from multiple sources. Attacks occur at different layers:

- **Layer 3/4 network attacks** like SYN floods or UDP amplification consume bandwidth or connection tables.
- **Layer 7 application attacks** send seemingly legitimate HTTP requests exhausting application resources.
- **DNS query floods** overwhelm DNS infrastructure.

Cloud providers have significant advantages in DDoS mitigation:

- **Massive scale**: AWS's global infrastructure absorbs enormous traffic volumes that would overwhelm individual organization's connections, distributed edge locations share attack traffic across many points of presence, and elastic scalability can auto-scale to handle attack traffic.
- **AWS Shield Standard** provides automatic protection included free, detecting and mitigating common attacks using traffic engineering and filtering, protecting infrastructure automatically without configuration, and absorbing attacks before they reach your applications.
- **AWS Shield Advanced** adds DDoS Response Team (DRT) 24/7 support, cost protection preventing scaling charges during attacks, enhanced detection and mitigation, integration with WAF for application-layer protection, and real-time attack visibility.
- **Architectural patterns**: CloudFront distributes traffic globally reducing attack concentration,

Route 53’s anycast network provides DNS DDoS resilience, ELB automatically distributes traffic across healthy instances, and auto-scaling handles traffic surges.

- **WAF** provides application-layer protection with rate limiting, geographic blocking, and request filtering.
- **Best practices**: avoid single points of failure, use managed services that handle DDoS automatically, implement rate limiting and traffic filtering, monitor for attack indicators, and have incident response plans.

Cloud providers' scale, expertise, and automated mitigation make them far better equipped to handle DDoS than individual organizations.

How do you ensure the security of data transferred between on-premises infrastructure and the cloud?

Secure hybrid connectivity requires encryption and access controls.

VPN connections:

- AWS Site-to-Site VPN creates encrypted tunnels over the internet using IPsec.
- Provides up to 1.25 Gbps per tunnel.
- Automatic failover with multiple tunnels.
- Integration with on-premises VPN devices.
- Suitable for moderate bandwidth needs and quick to establish.

AWS Direct Connect:

- Dedicated private connection between on-premises and AWS, bypassing public internet entirely.
- Provides consistent network performance.
- Reduced bandwidth costs for large transfers.
- Supports up to 100 Gbps.

For Direct Connect, I implement **encryption** using MACsec for Layer 2 encryption on supported connections or Site-to-Site VPN over Direct Connect for encryption in transit.

Transit Gateway centralizes connectivity for multiple VPCs and on-premises networks, simplifying management and enabling hub-and-spoke architectures.

Security controls:

- Implement strong encryption (IPsec with IKEv2, AES-256).
- Use private IP addressing preventing exposure.

- Implement BGP authentication preventing route injection.
- Enable CloudWatch metrics monitoring connection health.
- Use VPC Flow Logs monitoring traffic patterns.

Access control:

- Use security groups limiting what cloud resources on-premises can reach.
- Implement firewall rules on-premises controlling cloud access.
- Use PrivateLink for private access to AWS services.
- Apply least privilege to applications crossing boundaries.

Data protection:

- Encrypt sensitive data at application layer before transit (belt-and-suspenders).
- Implement certificate-based authentication.
- Use AWS Certificate Manager Private CA for internal PKI.
- Validate data integrity.

Monitoring:

- Use VPC Flow Logs.
- Enable CloudTrail for all AWS API calls from on-premises.
- Implement SIEM correlating on-premises and cloud logs.
- Alert on unusual cross-boundary traffic.

For maximum security, I prefer Direct Connect with VPN for encryption, avoiding public internet entirely while maintaining strong encryption.

Imagine you are responsible for reviewing the security of AWS Lambda functions in your organization’s environment. You discover a Lambda function that has an SSRF (Server-Side Request Forgery) vulnerability, and specifically at `http://127.0.0.1:9001/2018-06-01/runtime/invocation/next`. Explain the potential security risks associated with this SSRF vulnerability and how you would recommend mitigating these risks.

This is a critical finding because that specific endpoint is the **Lambda Runtime API** used by custom Lambda runtimes. An SSRF vulnerability allowing attacker-controlled requests to `127.0.0.1:9001` enables several attacks.

Security risks:

- The attacker could retrieve invocation events that might contain sensitive data (customer PII, credentials, API keys passed as event data).
- Manipulate function behavior by posting responses to invocation endpoints potentially causing the function to execute malicious logic.
- Retrieve environment variables via the runtime API which often contain secrets.
- Access the Lambda execution role’s temporary credentials from IMDSv2 at 169.254.170.2 (accessible from Lambda’s network namespace) giving them the function’s full IAM permissions.
- Potentially cause denial of service by interfering with the Lambda execution lifecycle.

This is especially dangerous because Lambda functions often have elevated permissions for AWS service access—compromising the function means assuming those permissions.

Mitigation recommendations:

- **Fix the SSRF vulnerability** through:
    - Strict input validation allow-listing expected URLs.
    - Implementing URL parsing that blocks localhost, 127.0.0.1, 169.254.x.x, and other private ranges.

    - Using application-layer controls preventing internal network access.
    - Conducting code review for all user-supplied URLs in requests.
- **Defense in depth**:
    - Implement least privilege IAM roles for the function granting only minimum necessary permissions.
    - Use resource-based policies on accessed resources adding additional authorization layer.
    - Avoid passing sensitive data in Lambda events—use secure parameter references instead.
    - Encrypt environment variables with KMS and rotate regularly.
    - Deploy Lambda in VPC with restrictive security groups if it needs VPC resources.
    - Implement WAF if Lambda is behind API Gateway filtering malicious requests.
    - Enable comprehensive logging with CloudWatch Logs and X-Ray.
- **Detection**:
    - Monitor CloudTrail for unusual API calls from the function's role.
    - Implement runtime application self-protection (RASP) detecting exploitation attempts.
    - Set up alerts on function errors or timeouts.
    - Use GuardDuty detecting anomalous behavior.

This vulnerability requires immediate remediation given the sensitive runtime API exposure and potential for full function compromise.

Have you worked on AWS WAF/Azure/GCP Cloud Armor. How will you test & implement core rule set in production. Provide the strategy.

Yes, I've implemented AWS WAF extensively. My strategy for testing and implementing core rule sets in production is phased and risk-averse.

Phase 1: Pre-Production Testing:

- Start by deploying WAF in a non-production environment mirroring production.
- Enable AWS Managed Rules Core Rule Set (CRS) and other relevant rule groups in **COUNT mode** (logging only, not blocking).
- Generate synthetic traffic including normal application flows and simulated attacks using tools like OWASP ZAP or known attack payloads.
- Analyze sampled requests and WAF logs to identify false positives where legitimate traffic would be blocked.

- Tune rules by creating custom rules with lower priority or excluding specific rule IDs for known false positives.

Phase 2: Production Deployment (Count Mode):

- Deploy WAF to production ALB, CloudFront, or API Gateway in COUNT mode.
- Enable comprehensive logging to S3 or Kinesis Firehose.
- Monitor for 1-2 weeks analyzing real production traffic patterns.
- Identify false positives through application logs correlating errors with WAF logs.
- Create exceptions for legitimate traffic (IP allowlists, custom rules).
- Document all tuning decisions with business justification.

Phase 3: Gradual Blocking:

- Switch specific high-confidence rule groups to BLOCK mode (SQL injection, XSS) while keeping others in COUNT.
- Implement detailed monitoring and alerting on blocked requests.
- Establish rapid rollback procedures if issues arise.
- Have on-call engineers ready during business hours.

Phase 4: Full Blocking:

- After validating initial rules, progressively enable additional rule groups.
- Switch remaining rules from COUNT to BLOCK.
- Maintain COUNT mode for new rules when added.
- Conduct regular reviews of blocked traffic ensuring no legitimate users affected.

Ongoing Operations:

- Weekly review of WAF metrics and sampled requests.
- Automated alerting on spikes in blocked requests indicating attacks or false positives.
- Quarterly tuning sessions reviewing all exceptions.
- Integration with incident response for attack analysis.
- Cost optimization reviewing rule usage and logs.

Key practices:

- Never enable blocking in production without count-mode testing.
- Maintain comprehensive logging for troubleshooting.
- Have rollback plan for each change.
- Use AWS WAF Security Automations for automated IP reputation lists and rate limiting.
- Treat WAF configuration as code with version control and peer review.

Explain AWS S3 buckets ransomware attacks, and what best practices would you recommend?

S3 ransomware attacks involve attackers encrypting or deleting objects in S3 buckets, then demanding ransom for recovery. The attack typically follows this pattern:

1. Attacker compromises AWS credentials (exposed access keys, compromised IAM user/role, SSRF vulnerability).
2. Validates access and identifies valuable S3 buckets containing critical data.
3. Exfiltrates data to external location for double-extortion leverage.
4. Encrypts objects using S3's server-side encryption or by downloading, encrypting locally, and re-uploading.
5. Or simply deletes objects if versioning isn't enabled or MFA Delete isn't configured.
6. Demands ransom threatening data exposure or permanent loss.

Best practices to prevent and mitigate:

Access control:

- Implement least privilege IAM policies.
- Use SCPs to prevent high-risk actions organization-wide.
- Require MFA for privileged operations.
- Regularly audit IAM permissions removing unnecessary access.
- Use IAM Access Analyzer to detect overly permissive policies.
- Implement cross-account access controls for backup buckets.

Versioning and protection:

- Enable S3 Versioning on all critical buckets preventing permanent deletion.
- Implement Object Lock in compliance mode making objects immutable for specified retention period (attackers can't delete even with admin access).
- Enable MFA Delete requiring MFA to delete versions or disable versioning.
- Use S3 Intelligent-Tiering to reduce costs while maintaining versions.

Backup and replication:

- Implement Cross-Region Replication to separate AWS account that attacker can't access.
- Use AWS Backup for centralized backup management with separate IAM permissions.
- Maintain offline backups or air-gapped copies for critical data.

- Regularly test restoration procedures.

Monitoring and detection:

- Enable CloudTrail data events for S3 tracking all object-level operations.
- Use EventBridge to alert on mass deletions or modifications.
- Implement GuardDuty S3 Protection detecting suspicious access patterns.
- Monitor for unusual API activity (bulk operations, access from new locations).
- Set up CloudWatch alarms on S3 metrics (request counts, error rates).

Encryption:

- Use customer-managed KMS keys with strict key policies.
- Implement separate KMS keys for different data classifications.
- Enable CloudTrail logging for all KMS operations.
- Use key policies preventing encryption with customer keys by unauthorized principals.

Network isolation:

- Use VPC endpoints for S3 access keeping traffic private.
- Implement bucket policies requiring access through specific VPC endpoints.
- Restrict public access using S3 Block Public Access at account and bucket levels.

Describe the steps you would take to detect and respond to a ransomware attack on an S3 bucket in real-time.

Detection mechanisms:

- Implement CloudWatch Events/EventBridge rules monitoring for specific patterns:
    - Multiple `DeleteObject` or `PutObject` events in short timeframe indicating bulk operations.
    - `PutBucketEncryption` or `PutBucketVersioning` changes that might disable protections.
    - Successful API calls from unusual geographic locations or IP addresses.
    - API calls using compromised credentials identified by GuardDuty.
- Enable GuardDuty S3 Protection detecting anomalous data access patterns, exfiltration attempts, and credential compromise.
- Set CloudWatch alarms on S3 metrics for abnormal request rates or error spikes.
- Implement custom Lambda functions analyzing CloudTrail logs in near-real-time for suspicious patterns like rapid sequential object modifications.

Real-time response workflow:

Step 1: Immediate containment:

- EventBridge rule detects suspicious activity and triggers Step Functions workflow.
- Lambda function immediately snapshots current bucket state documenting attack progression.
- Automated response modifies IAM policies or SCPs denying further S3 access to suspected compromised credentials/roles.
- If attack is confirmed, Lambda applies bucket policy denying all PutObject and DeleteObject operations temporarily (preserving existing data).

Step 2: Credential handling:

- Identify compromised credentials from CloudTrail `userIdentity` field.
- Immediately rotate or delete access keys if IAM user credentials.
- Revoke STS temporary credentials by modifying role trust policy if role assumed.
- Force all users to re-authenticate if widespread compromise suspected.
- Document all credentials used during attack window for forensic analysis.

Step 3: Assessment:

- Lambda function queries S3 versioning listing deleted or modified objects.
- Compares current object versions with previous state captured in compliance scans.
- Identifies scope of impact (how many objects affected, data sensitivity).
- Exports CloudTrail logs for the attack timeframe to secure forensic bucket.
- Generates initial incident report with timeline and affected resources.

Step 4: Communication:

- SNS notification alerts security team with incident severity and preliminary details.
- Create incident ticket in ServiceNow or PagerDuty with automated context.
- Notify data owners and compliance team based on affected data classification.
- Prepare communication templates for potential customer notification if PII affected.

Step 5: Recovery:

- If versioning enabled, Lambda function can automatically restore objects to previous versions before attack.
- If Object Lock enabled, immutable versions remain intact simplifying recovery.
- If cross-region replication configured, fail over to replica bucket.
- Validate restored data integrity through checksums or sample verification.

Step 6: Forensics:

- Preserve all logs (CloudTrail, VPC Flow Logs, application logs) in immutable storage.
- Analyze attacker's actions identifying initial access vector and lateral movement.
- Determine if data was exfiltrated (large data transfers, unusual network traffic).
- Document complete attack timeline.

Post-incident:

- Conduct root cause analysis identifying how credentials were compromised.
- Implement additional controls preventing recurrence.
- Update detection rules based on attack TTPs.
- Test recovery procedures.
- Share lessons learned.

Automation example: I'd implement this as infrastructure-as-code with EventBridge patterns detecting anomalies, Step Functions orchestrating response, Lambda functions executing containment actions, and SNS/Systems Manager handling notifications. The key is pre-built automation executing faster than attackers can complete encryption/deletion.

How cloud ransomware uses KMS to encrypt objects within Amazon S3 buckets of a compromised AWS account.

This is an insidious attack because it leverages AWS's own encryption mechanisms. When attackers compromise AWS credentials with sufficient S3 and KMS permissions, they can use KMS to encrypt S3 objects making recovery difficult without the attacker's cooperation.

Attack mechanism:

1. Attacker with compromised credentials that have `s3:PutObject` and `kms:GenerateDataKey` permissions creates a new KMS customer-managed key or uses existing key they have access to.
2. Downloads objects from target S3 bucket.
3. Encrypts them locally or uses S3's `PUT` operation with server-side encryption specifying `SSE-KMS` with their controlled KMS key.
4. Uploads encrypted versions overwriting original objects (if versioning disabled) or creating new encrypted versions.
5. Optionally deletes previous unencrypted versions if they have delete permissions.
6. Alternatively, they might use `CopyObject` with encryption parameters changing encryption from unencrypted or AWS-managed to their customer-managed key.
7. After encryption, attacker either deletes the KMS key (scheduling deletion) or rotates it, rendering objects undecryptable, or retains the key and demands ransom to provide decryption

access.

Why this is effective:

- Encrypted data appears normal in S3 - objects exist and aren't deleted, so basic monitoring might miss the attack.
- Without the KMS key, data is irrecoverable even for AWS support.
- If versioning isn't enabled, original unencrypted versions are lost.
- Even with versioning, if attacker deletes previous versions, recovery is impossible.

Prevention strategies:

KMS key policies:

- Implement strict key policies allowing encryption/decryption only by specific, necessary roles.
- Use condition statements requiring encryption with specific approved keys only.
- Deny `kms:ScheduleKeyDeletion` and `kms:DisableKey` for non-administrative users.
- Require MFA for key administrative actions.
- Use separate KMS keys for different data classifications with different permission sets.

S3 bucket policies:

- Require encryption with specific KMS keys using bucket policy conditions: `"s3:x-amz-server-side-encryption-aws-kms-key-id": "arn:aws:kms:region:account:key/key-id"`.
- Deny `PutObject` without proper encryption headers.
- Implement bucket policies preventing encryption with unauthorized keys.

Monitoring:

- Enable CloudTrail logging for all KMS operations alerting on `CreateKey`, `ScheduleKeyDeletion`, `DisableKey`, GenerateDataKey from unusual sources.
- Monitor S3 CloudTrail data events for encryption-changing operations (`CopyObject` with different encryption, `PutObject` with new SSE parameters).
- Use EventBridge to alert on KMS key policy modifications.
- Implement anomaly detection for unusual patterns of `GenerateDataKey` calls.

Access control:

- Apply least privilege limiting which roles can perform KMS operations.
- Use SCPs to prevent KMS key deletion or disabling across organization.
- Implement permission boundaries.
- Separate encryption permissions from data access permissions.

Backup and versioning:

- Enable S3 Versioning with MFA Delete.
- Maintain unencrypted backups (or encrypted with different keys) in separate account.
- Implement Object Lock preventing version deletion.
- Cross-region replication to isolated account.

Detection and response: Alert on bulk encryption operations, changes to object encryption metadata, new KMS keys created unexpectedly, or attempts to schedule key deletion. Automated response should block suspected compromised credentials immediately and preserve object versions.

Explain how you would implement versioning and lifecycle policies to prevent data loss in the event of a ransomware attack on S3.

Versioning and lifecycle policies create resilient S3 architecture protecting against ransomware and accidental deletion.

Versioning implementation:

- Enable S3 Versioning on all buckets containing important data - this maintains every version of every object, protecting against overwrites and deletions.
- When versioning is enabled, deleting an object creates a delete marker (the object appears deleted but versions remain).
- Overwriting an object creates a new version (previous version preserved).
- All versions can be restored.

Critical enhancement - MFA Delete:

- Enable MFA Delete requiring multi-factor authentication to permanently delete versions or disable versioning - this prevents attackers from destroying versions even with compromised credentials.
- Configure using:

  ```
  aws s3api put-bucket-versioning --bucket BUCKET --versioning-configuration
  Status=Enabled,MFADelete=Enabled --mfa "SERIAL TOKEN"
  ```

- Only the root account user can enable MFA Delete, adding protection.

Lifecycle policies for cost management:

- Versioning can increase storage costs dramatically.
- Implement lifecycle policies balancing protection and cost:
  - Transition noncurrent versions to cheaper storage classes (Glacier after 30 days, Deep Archive after 90 days).
  - Permanently delete noncurrent versions only after extended retention (365+ days for critical data).
  - Use Intelligent-Tiering for current versions.

Example policy:

```
{
  "Rules": [{
    "Id": "Archive old versions",
    "Status": "Enabled",
    "NoncurrentVersionTransitions": [
      {"NoncurrentDays": 30, "StorageClass": "GLACIER"},
      {"NoncurrentDays": 90, "StorageClass": "DEEP_ARCHIVE"}
    ],
    "NoncurrentVersionExpiration": {"NoncurrentDays": 365}
  }]
}
```

Object Lock for immutability:

- For compliance or critical data, implement S3 Object Lock in compliance mode with retention periods - objects become immutable and cannot be deleted or modified by anyone including root account until retention expires.
- This defeats ransomware completely for locked objects.
- Configure with retention period matching compliance requirements:

  ```
  aws s3api put-object-lock-configuration --bucket BUCKET --object-lock-configuration
  '{"ObjectLockEnabled": "Enabled", "Rule": {"DefaultRetention": {"Mode":
  "COMPLIANCE", "Days": 90}}}'
  ```

Cross-Region Replication with versioning:

- Configure CRR replicating all versions to bucket in separate AWS account with different credentials - if primary account compromised, replicated versions remain safe.
- Enable delete marker replication and version replication ensuring complete copy.

Monitoring version health:

- CloudWatch metrics tracking version counts detecting unusual spikes indicating mass overwrites.
- EventBridge rules alerting on version deletion attempts.

- Config rules ensuring versioning remains enabled.
- Regular audits confirming MFA Delete remains active.

Recovery procedures:

- Document process for restoring from versions.
- Test restoration regularly.
- Maintain automation for bulk version recovery.
- Ensure team knows how to identify correct version to restore.

Testing: Regularly simulate ransomware attack by deliberately encrypting test objects and practicing version-based recovery to validate protection works.

This multi-layered approach means even if attackers encrypt objects, previous versions remain recoverable, providing strong ransomware resilience.

What strategies and tools would you use to ensure consistent security across AWS, GCP, and Azure?

Multi-cloud security requires unified strategy despite platform differences.

Centralized identity:

- Implement single identity provider (Okta, Azure AD, Google Workspace) federating to all clouds via SAML/OIDC.
- Enforce MFA universally across all platforms.
- Use RBAC or ABAC with consistent role definitions mapped to cloud-specific permissions.
- Implement just-in-time access with approval workflows.
- Maintain centralized user lifecycle management.

Unified policy framework:

- Develop cloud-agnostic security policies (network isolation, encryption, logging, access control) mapping to cloud-specific implementations.
- Use policy-as-code with tools like OPA or HashiCorp Sentinel that work across clouds.
- Maintain security baselines as IaC templates per cloud (Terraform modules, CloudFormation, ARM templates).
- Document standard patterns for common architectures.

CSPM for continuous compliance:

- Deploy Cloud Security Posture Management tools with multi-cloud support - Prisma Cloud, Wiz, Orca Security, or Aqua providing unified visibility across AWS, GCP, Azure.
- Detect misconfigurations against common benchmarks (CIS).
- Identify compliance violations.
- Enable automated remediation.

Centralized logging and SIEM:

- Aggregate logs from all clouds into unified SIEM (Splunk, Sumo Logic, Elastic).
- Collect cloud audit logs (CloudTrail, Cloud Audit Logs, Activity Log).
- Normalize log formats for consistent querying.
- Implement correlation rules detecting cross-cloud attacks.
- Maintain single incident response workflow.

Network security:

- Implement consistent network segmentation principles across clouds.
- Use cloud interconnects (AWS Transit Gateway, Azure Virtual WAN, GCP Cloud Router) for secure inter-cloud communication.
- Deploy unified firewall policies via cloud-native firewalls or third-party NGFWs.
- Implement zero-trust networking with encryption everywhere.

Workload protection:

- Deploy cloud workload protection platforms (CWPP) like Lacework, Sysdig, or Aqua for container and serverless security.
- Implement runtime protection and vulnerability scanning consistently.
- Use service mesh (Istio, Consul) for consistent service-to-service authentication across clouds.
- Maintain common container security standards (image scanning, registry security).

Secrets management:

- Use unified secret manager (HashiCorp Vault, CyberArk) working across clouds, or cloud-native with documented synchronization (AWS Secrets Manager, GCP Secret Manager, Azure Key Vault).
- Implement consistent encryption key management.
- Use short-lived credentials everywhere.

Automation and IaC:

- Terraform or Pulumi for infrastructure across all clouds.
- Security scanning in CI/CD regardless of target cloud (Checkov, tfsec, Snyk IaC).
- Common deployment pipelines with security gates.

Governance:

- Tag resources consistently across clouds for ownership, cost allocation, and compliance.
- Use organizational hierarchy (AWS Organizations, GCP Organization, Azure Management Groups) with consistent policies.
- Implement billing and cost anomaly detection.
- Regular cross-cloud security reviews.

Challenges:

- Platform-specific features don't translate directly requiring mapping exercises.
- Different pricing models affecting cost of security controls.
- Varying maturity of security services requiring compensating controls.
- Complexity of managing multiple consoles requiring automation.

Tools: Prisma Cloud for CSPM, Terraform for IaC, Splunk for SIEM, and Okta for identity create strong multi-cloud security foundation.

How would you prevent such misconfigurations in the future?

Scenario: A development team deployed a database server in production VPC with security group allowing PostgreSQL (port 5432) from 0.0.0.0/0 for testing convenience. They intended to restrict it but forgot before going home. That night, automated scanners discovered the exposed database, attackers brute-forced weak database credentials (default postgres user with common password), exfiltrated customer data including PII, installed cryptocurrency miners consuming compute resources, and created backdoor database accounts for persistent access. The breach went undetected for days until customers reported unauthorized transactions.

How it happened:

- No code review for infrastructure changes.
- Lack of automated scanning detecting exposure.
- Absence of database activity monitoring missing abnormal queries.
- Weak authentication (no IAM database authentication).
- No network segmentation (database in public-facing subnet).

Prevention strategies:

Preventive controls:

- Implement IaC with security groups defined in Terraform reviewed before deployment.

- Policy-as-code (Sentinel, OPA) blocking security groups with 0.0.0.0/0 on sensitive ports during `terraform plan`.
- Security group templates with secure defaults.
- AWS Service Catalog providing pre-approved, secure configurations.
- Use SCPs preventing creation of overly permissive rules organization-wide:

```
{
"Effect": "Deny",
"Action": ["ec2:AuthorizeSecurityGroupIngress",
"ec2:AuthorizeSecurityGroupEgress"],
"Resource": "*",
"Condition": {"IpAddress": {"aws:SourceIp": "0.0.0.0/0"}}
}
```

Detective controls:

- AWS Config rules continuously checking for unrestricted security groups (AWS managed rule `restricted-common-ports` or custom rule for application-specific ports).
- Security Hub detecting exposed resources.
- GuardDuty identifying port scanning or brute force attempts.
- Automated scanning tools (Prowler, Scout Suite) in CI/CD and scheduled runs.
- Implement EventBridge rules alerting immediately on security group modifications: `{"source": ["aws.ec2"], "detail-type": ["AWS API Call via CloudTrail"], "detail": {"eventName": ["AuthorizeSecurityGroupIngress"]}}` triggering Lambda analyzing new rules and alerting if suspicious.

Automated remediation:

- Config remediation actions automatically revoking unrestricted rules.
- Lambda functions triggered by EventBridge removing problematic rules and notifying teams, balancing automation with preventing disruption.

Architecture:

- Never place databases in public subnets.
- Use private subnets with no internet route.
- Access via bastion hosts or Session Manager.
- Implement network ACLs as additional protection.
- Use VPC endpoints for AWS service access.

Additional controls:

- IAM database authentication eliminating password-based access.
- Database activity monitoring (CloudWatch Logs, native PostgreSQL logs).

- Encryption at rest and in transit.
- Regular vulnerability scanning with Inspector.
- Least privilege database permissions.

Process improvements:

- Mandatory security review for all infrastructure changes.
- Security training for developers on secure configurations.
- Incident response procedures for exposed resources.
- Regular penetration testing.

Monitoring:

- Alert on new database connections from unexpected sources.
- Unusual query patterns.
- Database errors indicating attacks.
- Integrate with SIEM correlating network and database events.

This comprehensive approach creates defense in depth preventing single misconfigurations from causing breaches.

What is AWS segmentation, and why is it important for securing cloud environments?

AWS segmentation is the practice of dividing cloud infrastructure into isolated zones with controlled communication between them, implementing defense in depth and limiting blast radius of security incidents. Segmentation occurs at multiple levels.

Account-level segmentation:

- Using AWS Organizations with separate accounts for different environments (dev, staging, prod), business units, or data classifications creates strong security boundaries.
- Compromising one account doesn't provide automatic access to others.
- I use accounts for workload isolation, security tool centralization (logging, security scanning in dedicated security account), and blast radius limitation.
- SCPs enforce organizational policies across accounts.

Network segmentation via VPCs:

- Each VPC is isolated network - traffic doesn't flow between VPCs without explicit peering or Transit Gateway attachment.
- Within VPC, subnets provide additional segmentation:

    - Public subnets for internet-facing resources.
    - Private subnets for application tiers.
    - Isolated subnets for data tier with no internet access.
- Route tables control traffic flow between subnets.

Micro-segmentation with security groups:

- Security groups create instance-level isolation - each resource can have different security groups allowing precise traffic control.
- I use security group referencing where app tier security group allows traffic only from web tier security group (no IP management needed).
- Database security group allows traffic only from app tier security group.
- This creates application-layer segmentation preventing lateral movement.

Why it's critical:

- **Containment** - if attacker compromises web server, segmentation prevents direct access to databases requiring additional compromises.
- **Compliance** - many frameworks require network segmentation (PCI DSS requires cardholder data environment isolation).
- **Blast radius reduction** - incidents affect only the compromised segment rather than entire infrastructure.
- **Lateral movement prevention** - attackers can't easily pivot between systems.
- **Traffic inspection** - chokepoints between segments enable monitoring and filtering.
- **Defense in depth** - multiple security layers requiring multiple bypasses.

Implementation best practices:

- Default deny with explicit allows.
- Minimize cross-segment communication to necessary only.
- Implement different security controls per segment based on sensitivity.
- Monitor all cross-segment traffic.
- Regularly review segmentation effectiveness.

Segmentation is foundational security architecture making breach containment possible.

How can it be used to implement network segmentation?

VPC peering creates private, encrypted networking connection between two VPCs enabling them to communicate as if on the same network. Traffic between peered VPCs stays on AWS's private network, doesn't traverse public internet, is encrypted automatically, and has no single point of failure or bandwidth bottleneck.

Configuration:

1. Create peering connection between VPCs (can be in same or different accounts/regions).
2. Accept the peering request in target VPC.
3. Update route tables in both VPCs adding routes for peer VPC's CIDR blocks.
4. Configure security groups allowing traffic from peer VPC CIDR or security groups.

For network segmentation: VPC peering enables controlled connectivity between isolated VPCs. I use it to create segmented architecture where different workloads or environments reside in separate VPCs (production VPC, development VPC, shared services VPC for Active Directory or monitoring tools) with peering allowing necessary communication while maintaining isolation.

Security benefits:

- **Non-transitive routing** - if VPC A peers with VPC B, and VPC B peers with VPC C, VPC A cannot access VPC C unless explicitly peered. This prevents unintended access paths.
- **Granular control** - route tables in each VPC control which subnets can communicate with peer, security groups control traffic at instance level, and NACLs provide additional subnet-level filtering.
- **Isolated failure domains** - issues in one VPC don't affect others except for specific peered connections.
- **Audit trail** - VPC Flow Logs capture traffic between peered VPCs for security monitoring.

Use case example:

- Production VPC (10.0.0.0/16) hosts customer-facing applications.
- Shared services VPC (10.1.0.0/16) hosts centralized logging, monitoring, and Active Directory.
- Management VPC (10.2.0.0/16) hosts bastion hosts and administrative tools.
- Peering connections allow:
  - Production VPC to reach shared services for logging (specific route for 10.1.0.0/16).
  - Management VPC to reach production for administration (specific route for 10.0.0.0/16).
  - But production cannot directly reach management (no route) preventing compromised production resources from attacking management infrastructure.

Limitations: VPC peering doesn't scale to many VPCs (full mesh becomes complex - 100 VPCs needs 4,950 peering connections). For larger environments, Transit Gateway provides hub-and-spoke topology simplifying management.

Best practices:

- Use peering for few VPCs with specific connectivity requirements.

- Implement strict security group rules even between peers.
- Monitor cross-VPC traffic with Flow Logs.
- Document peering relationships and their purposes.
- Regularly review whether peering is still necessary.

VPC peering enables flexible segmentation while maintaining control.

How do you configure security groups and network ACLs to enforce network segmentation within an AWS VPC?

Security groups and NACLs work together for defense in depth in network segmentation.

Security group strategy: I design tier-based security groups aligned with application architecture - web tier, application tier, and data tier.

- **Web tier security group**: allows inbound HTTPS (443) from 0.0.0.0/0 for public access and SSH (22) from management security group only for administration.
- **Application tier security group**: allows inbound 8080 or application port from web tier security group (using security group ID as source, not CIDR), no direct internet access, and SSH from management security group.
- **Database tier security group**: allows inbound 3306 (MySQL) or 5432 (PostgreSQL) from application tier security group only, denies all other inbound traffic, and SSH/Session Manager from management security group for administration.

This creates **micro-segmentation** where database only accepts connections from application tier, and application only from web tier. Using security group IDs as sources instead of CIDR ranges means adding instances to tiers doesn’t require security group updates.

NACL strategy: NACLs provide subnet-level protection complementing security groups. I use them more sparingly since security groups handle most traffic control.

- **Public subnet NACL**:
  - Explicitly allows inbound 443 and 80 from 0.0.0.0/0.
  - Allows outbound ephemeral ports (1024-65535) for response traffic.
  - Denies known malicious IP ranges (threat intelligence feeds).
  - Allows VPC CIDR for internal communication.
- **Private subnet NACL**:
  - Denies direct inbound from internet.
  - Allows inbound from VPC CIDR ranges.

    - Allows outbound to internet for updates via NAT gateway.
    - Blocks inbound on administrative ports (22, 3389) except from specific management subnet.
- **Database subnet NACL**:
    - Denies all inbound except from application subnet CIDR on database ports.
    - Denies all outbound except responses.
    - Provides additional protection against compromised instances scanning internally.

Implementation approach:

- Start with deny-all NACLs and security groups.
- Add only necessary allow rules based on application communication requirements.
- Use security group references instead of IP addresses wherever possible for maintainability.
- Implement rule naming conventions describing purpose.
- Document exception requests.

Monitoring and validation:

- Enable VPC Flow Logs at subnet level capturing all traffic.
- Analyze flows for unexpected connections indicating misconfiguration or compromise.
- Use GuardDuty detecting anomalous network behavior.
- Implement automated testing trying to connect between tiers that shouldn't communicate.
- Regular architecture reviews ensuring segmentation matches design.

Automation:

- Define security groups and NACLs as IaC (Terraform, CloudFormation) with code review required for changes.
- Implement policy-as-code preventing overly permissive rules.
- Use AWS Config rules detecting non-compliant configurations.

Example flow: Internet user connects to web server (allowed by web SG), web server connects to app server (allowed because web SG is source in app SG rule), app server connects to database (allowed because app SG is source in DB SG rule). If attacker compromises web server and tries directly accessing database, connection fails because web SG isn't allowed in DB SG - forcing attacker through multiple layers.

This layered approach with security groups providing granular instance-level control and NACLs providing subnet-level boundaries creates robust network segmentation.

Describe the benefits and use cases of using AWS Transit Gateway for network segmentation.

Transit Gateway acts as cloud router enabling VPCs, VPN connections, and Direct Connect to interconnect through hub-and-spoke topology instead of complex mesh.

Benefits for segmentation:

- **Simplified connectivity** - instead of managing hundreds of VPC peering connections in full mesh (n*(n-1)/2 connections), Transit Gateway provides central hub requiring only single attachment per VPC (n connections). With 50 VPCs, this reduces from 1,225 peering connections to 50 attachments.
- **Centralized routing control** - Transit Gateway route tables define which VPCs can communicate, enabling creation of isolated routing domains. I can have production route table where prod VPCs communicate, development route table for dev VPCs, and shared services route table accessible by both, all on same Transit Gateway.
- **Network segmentation patterns**: Route table associations determine which VPCs can reach each other implementing segmentation at scale.
- **Scalability** - supports thousands of VPCs and high bandwidth (up to 50 Gbps per VPC attachment, bursts higher), scales way beyond VPC peering limitations.
- **Multi-account and multi-region** - Transit Gateway can be shared across AWS accounts via Resource Access Manager, and Transit Gateway peering connects Transit Gateways across regions enabling global network architecture.

Use cases:

- **Enterprise hub-and-spoke** - centralized shared services VPC (Active Directory, DNS, monitoring) accessible from all spoke VPCs, while spoke VPCs remain isolated from each other. Shared services attached to one route table, spokes attached to their own isolated route tables with routes only to shared services.
- **Inspection architecture** - all inter-VPC traffic routes through security VPC containing firewalls, IDS/IPS, or traffic inspection appliances. Transit Gateway routes traffic through inspection VPC before reaching destination enabling centralized security controls.
- **Isolated environments with controlled access** - production, staging, and development environments in separate VPCs attached to Transit Gateway with production having no routes to dev/staging, but management VPC has routes to all for administration. This prevents accidental production impact from dev/test while maintaining admin access.
- **Hybrid cloud segmentation** - on-premises network connects via VPN or Direct Connect to Transit Gateway, with specific routes allowing access only to designated VPCs (like DMZ VPC) while blocking direct access to internal workload VPCs.
- **Multi-region architecture** - Transit Gateway in each region handling intra-region connectivity, with Transit Gateway peering providing inter-region communication, enabling global

segmentation with regional isolation.

Security benefits:

- Centralized network monitoring with VPC Flow Logs from Transit Gateway attachments.
- Chokepoint for implementing security controls (route through inspection VPC).
- Network policy enforcement through route tables preventing unauthorized communication.
- Audit trail via CloudTrail logging all Transit Gateway configuration changes.
- DDoS protection as traffic flows through centralized paths with monitoring.

Implementation considerations:

- Plan IP addressing carefully avoiding overlapping CIDRs across VPCs.
- Use route table tags and naming for clear segmentation purpose.
- Implement automation for attachment and route management.
- Monitor Transit Gateway metrics (bytes processed, packets dropped).
- Design for high availability using multiple availability zones.

Cost consideration: Transit Gateway has hourly charge per attachment plus data processing charges, so evaluate cost versus complexity for smaller deployments where VPC peering might be more economical.

For large-scale environments with complex segmentation needs, Transit Gateway provides superior manageability and security.

What are the key considerations when implementing cross-account access controls for AWS resources in a segmented environment?

Cross-account access requires careful security design balancing functionality and isolation.

- **IAM roles for cross-account access**: Preferred method over IAM users. Create role in target account (Account B) with permissions to access resources, configure trust policy allowing source account (Account A) to assume role specifying principal: `"Principal": {"AWS": "arn:aws:iam::ACCOUNT-A:root"}`, and users/roles in Account A need `sts:AssumeRole` permission to assume the role in Account B.
- **Trust policy security**: Never use `"Principal": {"AWS": "*"}` allowing any AWS account to attempt assumption - this is critical mistake. Specify exact account IDs, use `ExternalId` for third-party access preventing confused deputy problem where attacker tricks you into accessing their resources.

- **External ID pattern**: For scenarios where multiple customers use same role, require External ID: `"Condition": {"StringEquals": {"sts:ExternalId": "unique-external-id"}}`. This prevents customer A from assuming role intended for customer B.
- **Least privilege in cross-account**: Grant minimal permissions in target account role - don't give `AdministratorAccess` when specific S3 bucket access suffices. Use resource-based policies (S3 bucket policies, KMS key policies) as additional authorization layer requiring both IAM role permission AND resource policy permission for access.
- **Session tags and ABAC**: Use session tags during role assumption to pass context, implement attribute-based access control in target account using tags: `"Condition": {"StringEquals": {"aws:PrincipalTag/Project": "ProjectX"}}` limiting what assumed role can access based on attributes.
- **Monitoring and auditing**: Enable CloudTrail in all accounts logging `AssumeRole` calls showing who assumed which roles when, use CloudWatch Events detecting cross-account assumptions from unexpected sources, track resource access from external accounts with resource-level CloudTrail logging (S3 data events, Lambda invocations), and implement alerts on new cross-account trust relationships.
- **SCPs for guardrails**: Use Service Control Policies to prevent certain cross-account actions organization-wide, block resource sharing with external AWS accounts unless explicitly allowed, prevent assume role to accounts outside organization, and enforce requirement for External ID.
- **Resource-based policies**: S3 bucket policies, KMS key policies, SNS topic policies, and SQS queue policies explicitly define which external accounts can access. Use condition statements for additional controls: `"Condition": {"StringEquals": {"aws:SourceAccount": "ACCOUNT-A"}}` ensuring access only from specific account.
- **Secrets and encryption**: For cross-account S3 access with encryption, KMS key policy must allow external account to use key for decryption. Use separate KMS keys per account/environment, grant explicit cross-account access in key policies, and monitor key usage with CloudTrail.
- **Network considerations**: Cross-account VPC peering or Transit Gateway attachments for network connectivity, use PrivateLink for private cross-account service access avoiding internet, and implement security groups and NACLs controlling cross-account traffic.
- **Segmentation benefits**: Keep accounts isolated with cross-account access as explicit exception, implement different security controls per account (production has stricter controls than development), and maintain blast radius containment where compromise of one account doesn't automatically grant access to others.
- **Best practices**: Document all cross-account relationships with business justification, regularly review and audit cross-account access removing unnecessary permissions, use automation (CloudFormation StackSets, Terraform) for consistent cross-account role deployment, implement approval workflows for new cross-account access requests, and use AWS Organizations for centralized management.
- **Example scenario**: Account A (development) needs to copy AMIs to Account B (production) for deployment. Create role in Account B with permissions to create AMIs and copy snapshots, trust policy allowing Account A's specific CI/CD role to assume it, Account A's CI/CD role assumes Account B's role when copying AMIs, all assumptions logged in both accounts' CloudTrail, and

automated review quarterly ensuring access still needed.

What is the purpose of the IAM PassRole permission, and how is it used in AWS?

The `iam:PassRole` permission allows an IAM principal to pass an IAM role to an AWS service when creating or modifying resources. This is necessary because many AWS services need to assume roles to perform actions on your behalf.

For example:

- When creating an EC2 instance, you attach an instance profile (IAM role) that the instance will use for API calls. The user creating the instance needs `iam:PassRole` permission to assign that role.
- Creating a Lambda function requires passing an execution role to Lambda.
- Deploying a CloudFormation stack may pass roles to various services.
- Creating an ECS task requires passing a task execution role.

How it works: When you call an API like `ec2:RunInstances` with `IamInstanceProfile` parameter or `lambda:CreateFunction` with `Role` parameter, AWS checks two things:

1. Does the calling principal have permission to perform the service action (like `ec2:RunInstances`).
2. Does the calling principal have `iam:PassRole` permission for the specific role being passed.

Both must be true for the operation to succeed.

Purpose: This permission exists as a security boundary preventing privilege escalation. Without it, users with `ec2:RunInstances` permission could create instances with administrator roles, effectively gaining admin access through the instance. PassRole acts as a gate ensuring users can only assign roles they're explicitly permitted to pass, maintaining least privilege.

The permission structure is:

```
{
  "Effect": "Allow",
  "Action": "iam:PassRole",
  "Resource": "arn:aws:iam::ACCOUNT:role/ROLE-NAME"
}
```

The resource specifies which roles can be passed, and conditions can further restrict when/how they can be passed. This is foundational to AWS security architecture, separating the ability to create resources from the ability to grant those resources elevated permissions.

Explain the potential security risks associated with granting the PassRole permission to IAM roles.

PassRole permission creates significant privilege escalation risks if not carefully controlled.

Primary risk - privilege escalation:

- A user with `iam:PassRole` on a highly privileged role (like an administrator role) and permissions to create services (EC2, Lambda, CloudFormation) can escalate their privileges by creating resources with the privileged role, then using those resources to perform actions they couldn't do directly.
- For example: user with limited permissions has `iam:PassRole` on admin role and `lambda:CreateFunction`, creates Lambda function with admin role, invokes the Lambda function executing admin-level actions, effectively bypassing their permission restrictions.

Lateral movement: Attacker compromising an account with broad PassRole permissions can assume different roles in the environment, potentially accessing resources in different VPCs, accounts, or security domains, pivoting through the infrastructure using different role identities.

Confused deputy attack: If PassRole allows passing roles to external services or accounts without proper conditions, attackers could trick your services into performing actions on their behalf using your credentials.

Long-term persistence: Attacker with PassRole permission could create long-lived resources (EC2 instances, Lambda functions) with privileged roles providing persistent access even after initial compromise is remediated, or create CloudFormation stacks that recreate attack infrastructure if deleted.

Data exfiltration: Passing data-access roles to attacker-controlled services (Lambda writing to attacker's S3, EC2 instances in attacker's network) enables data theft.

Compliance violations: Uncontrolled PassRole can lead to resources with inappropriate permissions violating compliance requirements, or audit trail confusion where actions appear from service roles rather than user identities.

Resource-based policy bypass: PassRole combined with resource creation can bypass resource-based policies by creating resources that access others through assumed roles.

The fundamental issue is that PassRole separates identity permissions from resource permissions—a user with minimal direct permissions but broad PassRole can effectively have unlimited access through passed roles. This makes PassRole one of the most security-sensitive permissions requiring strict control.

How do you restrict the usage of the PassRole permission to specific roles and resources while ensuring security?

Restricting PassRole requires multiple layers of control.

Specific role ARNs: Never grant `iam:PassRole` with `Resource: "*"`. Always specify exact role ARNs that can be passed:

```
{
  "Effect": "Allow",
  "Action": "iam:PassRole",
  "Resource": [
    "arn:aws:iam::ACCOUNT:role/EC2-App-Role",
    "arn:aws:iam::ACCOUNT:role/Lambda-Processing-Role"
  ]
}
```

This limits which roles can be assigned to resources.

Service-specific conditions: Use the `iam:PassedToService` condition key restricting which AWS services can receive the role:

```
{
  "Effect": "Allow",
  "Action": "iam:PassRole",
  "Resource": "arn:aws:iam::ACCOUNT:role/Lambda-*",
  "Condition": {
    "StringEquals": {
      "iam:PassedToService": "lambda.amazonaws.com"
    }
  }
}
```

This prevents passing Lambda roles to EC2 or other services.

Role naming conventions: Implement strict naming standards for roles (like `Service-Environment-Purpose` pattern: `Lambda-Prod-DataProcessor`) and use wildcards in PassRole permissions based on naming: `Resource: "arn:aws:iam::ACCOUNT:role/Lambda-*"` allowing passing any Lambda role but not EC2 roles. Document the conventions and enforce through automation.

Resource tagging conditions: Tag roles with their intended purpose and use condition keys in PassRole permissions:

```
{
  "Condition": {
    "StringEquals": {
      "iam:ResourceTag/Environment": "Development"
    }
  }
}
```

allowing users to pass only development-tagged roles, preventing production role assignment.

Permission boundaries on passable roles: Implement permission boundaries on roles that can be passed, limiting maximum permissions even if someone passes them. If a role has boundary restricting it to specific S3 buckets, passing that role can't grant broader access.

Combining with service permissions: PassRole is useless without corresponding service permissions. Control both: grant `lambda:CreateFunction` and `iam:PassRole` for specific Lambda roles only, but don't grant `ec2:RunInstances` preventing Lambda role usage with EC2.

SCPs for organization-wide controls: Use Service Control Policies preventing PassRole of administrator or sensitive roles across entire organization:

```
{
  "Effect": "Deny",
  "Action": "iam:PassRole",
  "Resource": "arn:aws:iam::*:role/*Admin*"
}
```

Monitoring and detection: CloudTrail logs all PassRole operations—monitor for unexpected role passing, alert on PassRole of highly privileged roles, detect patterns indicating privilege escalation attempts (creating service resources immediately after PassRole), and use Access Analyzer to identify overly broad PassRole permissions.

Example restrictive policy:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "iam:PassRole",
    "Resource": "arn:aws:iam::123456789012:role/Lambda-Prod-*",
    "Condition": {
      "StringEquals": {
        "iam:PassedToService": "lambda.amazonaws.com"
      }
    }
  }]
}
```

This allows passing only production Lambda roles and only to Lambda service. This prevents both service confusion and role type confusion.

Describe a scenario where you would use the PassRole permission in AWS IAM, and how would you ensure its security?

Scenario: A DevOps team needs to deploy Lambda functions for data processing pipelines. These functions need to read from S3, write to DynamoDB, and publish to SNS. The team shouldn't have direct access to production data, but their Lambda functions need it.

Solution using PassRole:

1. Create Lambda execution role `Lambda-Prod-DataProcessor` with specific permissions: read from `data-input-*` S3 buckets, write to `DataProcessing` DynamoDB table, and publish to `processing-results` SNS topic.
2. This role has a trust policy allowing Lambda service to assume it: `{"Principal": {"Service": "lambda.amazonaws.com"}}`.
3. Create IAM group `DevOps-Lambda-Deployers` with permissions: `lambda:CreateFunction`, `lambda:UpdateFunctionConfiguration`, `iam:PassRole` for specific role with service restriction.

IAM policy for DevOps:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:UpdateFunctionCode",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "arn:aws:lambda:us-east-1:ACCOUNT:function:DataProcessing-*"
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "arn:aws:iam::ACCOUNT:role/Lambda-Prod-DataProcessor",
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "lambda.amazonaws.com"
        }
      }
    }
  ]
}
```

Security measures:

- The PassRole permission is scoped to single specific role, not wildcard roles.
- The condition ensures role can only be passed to Lambda, preventing EC2 or other service usage.
- The Lambda function name must match pattern `DataProcessing-*` preventing unrelated function creation.
- DevOps team members cannot directly read S3 or write DynamoDB—they can only create functions that can.
- Permission boundary on the Lambda role limits maximum permissions preventing escalation even if DevOps modifies role (they can't, lacking `iam:PutRolePolicy`).

Monitoring:

- CloudTrail alerts on PassRole operations for this role.
- Lambda function creation/updates logged and reviewed.
- Unusual invocations of Lambda functions detected through CloudWatch metrics.
- Quarterly access review ensuring PassRole is still necessary.

Separation of duties:

- DevOps deploys functions but can't modify the execution role's permissions (Security team manages role).
- Security team defines what functions can access but doesn't deploy code (DevOps deploys).
- Neither team has direct data access requiring collaboration.

Testing: Before production, test in development environment with similar role structure, verify DevOps cannot escalate privileges through the Lambda role, and confirm audit trails capture all role passage.

This scenario demonstrates proper PassRole usage enabling teams to deploy workloads without direct access to sensitive resources while maintaining security through scoping, conditions, and monitoring.

What best practices would you follow when managing IAM roles with PassRole permissions in a AWS environment?

Principle of least privilege:

- Grant PassRole only for specific roles required, not wildcard.
- Each user/group should be able to pass only roles necessary for their job function—developers deploying Lambda functions pass only Lambda execution roles, not EC2 instance roles.

Service-specific scoping: Always use `iam:PassedToService` condition:

```
"Condition": {
  "StringEquals": {
    "iam:PassedToService": ["lambda.amazonaws.com", "ecs-tasks.amazonaws.com"]
  }
}
```

preventing cross-service role abuse.

Role naming and organization:

- Implement consistent naming conventions enabling wildcard PassRole permissions that remain secure: `Lambda-{Environment}-{Purpose}`, `EC2-{Environment}-{Application}`.
- DevOps team gets `iam:PassRole` on `arn:aws:iam::*:role/Lambda-Dev-*` for development Lambda roles only.

Permission boundaries on passable roles: All roles that can be passed should have permission boundaries limiting maximum permissions. Even if someone unauthorized passes the role, boundary prevents escalation: `"PermissionsBoundary": "arn:aws:iam::ACCOUNT:policy/Lambda-Boundary"`.

Separation of role management: Teams with PassRole permissions should NOT have permissions to modify the roles they can pass (`iam:PutRolePolicy`, `iam:AttachRolePolicy`). Separate role management (Security team) from role usage (DevOps team).

Regular audits:

- Quarterly review of all PassRole permissions using IAM Access Analyzer.
- Identify overly broad permissions or unused PassRole grants.
- Verify passed roles still follow least privilege.

Automated detection:

- CloudTrail monitoring alerting on PassRole operations especially for sensitive roles.
- GuardDuty detecting privilege escalation attempts.
- Config rules ensuring PassRole permissions include proper conditions.

Documentation and training:

- Document which roles can be passed and why.
- Train teams on PassRole security implications and proper usage.
- Establish approval process for new PassRole permissions requiring security review.

SCPs for guardrails: Organization-level denies preventing PassRole of critical roles:

```
{
  "Effect": "Deny",
  "Action": "iam:PassRole",
  "Resource": [
    "arn:aws:iam::*:role/*Admin*",
    "arn:aws:iam::*:role/OrganizationAccountAccessRole"
  ]
}
```

Resource tagging: Tag passable roles with metadata (environment, team, purpose) and use tag conditions in PassRole permissions: `"Condition": {"StringEquals": {"iam:ResourceTag/Team": "DataEngineering"}}` ensuring teams only pass their own team's roles.

Avoid PassRole with modify permissions: Be extremely cautious granting PassRole to principals with `iam:UpdateAssumeRolePolicy` or role modification permissions—this combination enables trivial privilege escalation.

Monitoring role usage: Track not just PassRole operations but also what passed roles actually do—unusual API calls from Lambda execution roles might indicate compromised deployment process.

Emergency procedures: Have runbooks for suspected PassRole abuse including immediate revocation procedures, role assumption tracking, and forensic log preservation.

CloudFormation and IaC considerations: When using CloudFormation, deployer needs PassRole for roles in template—use CloudFormation service roles limiting what templates can deploy rather than giving developers broad PassRole.

These practices treat PassRole as the high-risk permission it is, implementing defense in depth around it.

What is the AWS CIS (Center for Internet Security) Benchmark, and why is it important for securing AWS resources?

The AWS CIS Benchmark is a consensus-based security configuration guide developed by cybersecurity experts worldwide defining best practices for securely configuring AWS environments. It provides specific, actionable recommendations across AWS services organized by security domains.

Purpose and importance:

- The benchmark establishes **industry-standard security baseline** recognized globally.
- Provides **prescriptive guidance** with specific configuration instructions not just general

principles.

- Enables **compliance framework mapping** as many regulations reference CIS benchmarks.
- Facilitates **audit preparation** by implementing controls auditors expect.
- Offers **risk reduction** by addressing common misconfigurations leading to breaches.

Structure: The benchmark is organized into sections covering Identity and Access Management (IAM), logging and monitoring, networking, compute (EC2), storage (S3), and other AWS services. Each recommendation has a profile level (Level 1 for basic security all organizations should implement, Level 2 for enhanced security for environments requiring additional protection) and includes rationale explaining why the control matters, audit procedures for checking compliance, and remediation steps for fixing non-compliance.

Example recommendations:

- Enable MFA for root account and all IAM users with console access.
- Ensure CloudTrail is enabled in all regions with log file validation.
- Eliminate root account access keys.
- Enforce strong password policies.
- Ensure S3 buckets have server access logging enabled.
- Ensure VPC flow logging is enabled for all VPCs.
- Avoid using root account for daily operations.

Why it's important:

- CIS benchmarks represent **collective expertise** from security professionals across industries.
- Provide **actionable guidance** unlike vague security advice.
- Are **regularly updated** to address new services and threats.
- Offer **measurable compliance** through automated scanning tools.
- Create **common language** for discussing security across organizations and with auditors.

Many organizations use CIS benchmarks as foundation for their security baseline, layering additional controls on top. For AWS specifically, the benchmark addresses cloud-specific risks that general security frameworks might miss. Implementing CIS benchmark recommendations significantly hardens AWS environments against common attack vectors and misconfigurations.

Describe some key security checks included in the AWS CIS Benchmark for AWS Identity and Access Management (IAM).

The IAM section of CIS Benchmark contains critical foundational controls:

Root account security:

- Avoid using root account for everyday tasks (1.1).
- Ensure MFA is enabled on root account (1.2).
- Ensure root account access keys don't exist (1.3 - root keys are extreme security risk and almost never necessary).
- Ensure no root account usage in last 30 days (tracking through credential report).

MFA enforcement:

- Ensure MFA is enabled for all IAM users with console passwords (1.4), preventing credential compromise from password theft alone.
- Implement MFA on privileged accounts and roles (additional protection for high-risk access).

Credential management:

- Ensure access keys are rotated every 90 days or less (1.5).
- Ensure IAM password policy requires minimum length of 14 characters (1.6).
- Password policy requires at least one uppercase letter (1.7).
- One lowercase letter (1.8).
- One number (1.9).
- One symbol (1.10).
- Prevents password reuse (1.11).
- Ensure unused credentials are disabled or removed within 90 days (1.12 - old credentials are security liability).

Least privilege:

- Ensure IAM policies that allow full ":" administrative privileges aren't attached to users (1.16 - admin access should be through roles with temporary credentials).
- Ensure IAM policies are attached only to groups or roles not users (1.15 - centralized management).
- Ensure credentials unused for 90 days are disabled (1.3 - reducing attack surface).

Access controls:

- Ensure no IAM policies allow full ":" administrative privileges (1.22).
- Ensure IAM users receive permissions only through groups (1.15).
- Maintain a support role for AWS support case management (1.17).
- Ensure IAM instance roles are used for AWS resource access from instances (1.19 - not hardcoded credentials).

Hardware MFA for root: Ensure hardware MFA is enabled for root account (1.13 - more secure than virtual MFA for highest-privilege account).

Password policy:

- Ensure password policy expires passwords within 90 days or less (1.11).
- Ensure password policy prevents password reuse (maintains password history).

These controls address the most common IAM misconfigurations leading to account compromise. Implementing them creates strong identity security foundation. The emphasis on root account protection, MFA, credential lifecycle management, and least privilege reflects real-world attack patterns where compromised credentials and excessive permissions enable most cloud breaches.

How do you use AWS Config to check compliance with the AWS CIS Benchmark, and what actions would you take if non-compliance is detected?

AWS Config continuously monitors and records AWS resource configurations enabling automated CIS Benchmark compliance checking.

Implementation:

- Enable AWS Config in all regions recording all resource types.
- Configure Config to deliver configuration snapshots and history to centralized S3 bucket in security account.
- Set up Config Aggregator collecting configuration data from multiple accounts and regions into single view.
- Enable Config rules mapped to CIS Benchmark recommendations.

Config rules for CIS Benchmark: AWS provides managed Config rules matching many CIS controls:

- `root-account-mfa-enabled` checks CIS 1.2.
- `iam-user-mfa-enabled` checks CIS 1.4.
- `access-keys-rotated` checks CIS 1.5.
- `iam-password-policy` checks CIS 1.6-1.11.
- `cloudtrail-enabled` checks logging requirements.
- `cloud-trail-log-file-validation-enabled` checks log integrity.

For controls without managed rules, create custom Config rules using Lambda functions—for example, checking root account hasn't been used in 30 days requires custom rule querying credential reports.

Conformance packs: AWS offers CIS Benchmark conformance packs bundling all related Config rules into single deployment. Deploy with:

```
aws configservice put-conformance-pack --conformance-pack-name cis-aws-foundations-
benchmark --template-s3-uri s3://bucket/cis-template.yaml
```

This enables all CIS rules at once with proper configurations.

When non-compliance detected: Config marks resources as non-compliant and generates findings.

Immediate response:

- For critical violations (root access keys exist, MFA disabled on root), trigger automated remediation through Config Remediation Actions or EventBridge invoking Lambda—for example, Lambda function sends high-priority alert to security team and creates P1 incident ticket.
- For root access keys, manual intervention required but automation escalates immediately.

Investigation:

- Review configuration timeline in Config showing when resource became non-compliant and what changed.
- Check CloudTrail for API calls causing non-compliance.
- Identify who/what made the change.

Remediation:

- For automatable fixes, enable Config auto-remediation—IAM password policy violations automatically corrected by applying compliant policy, S3 public access blocks enabled automatically, and unencrypted volumes encrypted.
- For issues requiring human judgment (unused IAM users), create tickets assigned to resource owners with SLA based on risk.

Tracking:

- Use Config Compliance Dashboard viewing organization-wide compliance status.
- Trend analysis showing improvement or degradation over time.
- Security Hub integration aggregating Config findings with other security tools.

Reporting:

- Generate compliance reports for audits showing configuration at specific times.
- Automated weekly reports to leadership on compliance posture.
- Exception tracking documenting approved deviations from benchmark.

Continuous improvement:

- Quarterly review of non-compliant resources identifying systemic issues.
- Update IaC templates to deploy compliant configurations by default.
- Refine custom Config rules based on false positives.

Prevention: Once baseline compliance achieved, use Config rules in proactive mode preventing non-compliant resource creation, integrate with CI/CD preventing deployment of non-compliant infrastructure, and implement SCPs enforcing organization-wide compliance.

The key is treating Config compliance as continuous process not point-in-time audit, with automation for detection, escalation, and remediation where appropriate.

Explain the importance of enabling AWS CloudTrail and AWS Config to align with the CIS Benchmark requirements.

CloudTrail and Config are foundational services required by multiple CIS Benchmark controls and essential for security visibility.

CloudTrail importance: CIS Benchmark section 2 (Logging) mandates CloudTrail because it provides **comprehensive audit trail** recording all API calls made in AWS account—who did what, when, from where, and what the result was. This is critical for:

- **Security investigations** enabling incident response teams to trace attacker actions.
- **Compliance requirements** as most frameworks require audit logging.
- **Governance** understanding how infrastructure changes over time.
- **Anomaly detection** establishing baselines and identifying suspicious activity.

Specific CIS requirements:

- Ensure CloudTrail is enabled in all regions (2.1 - attacks might occur in unexpected regions).
- Ensure CloudTrail log file validation is enabled (2.2 - cryptographic validation prevents log tampering).
- Ensure S3 bucket used for CloudTrail logs is not publicly accessible (2.3).
- Ensure CloudTrail logs are integrated with CloudWatch Logs (2.4 - enabling real-time monitoring and alerting).
- Ensure S3 bucket access logging is enabled for CloudTrail bucket (2.6 - meta-logging for security).
- Ensure CloudTrail logs are encrypted at rest using KMS CMKs (2.7 - protecting sensitive log data).
- Ensure rotation is enabled for KMS CMKs encrypting CloudTrail (2.8).

CloudTrail without these controls is partially effective but has gaps—without multi-region, attacks in unusual regions go undetected; without log file validation, attackers can tamper with evidence; without encryption, log data might be exposed; and without CloudWatch integration, detection is delayed requiring batch log analysis.

AWS Config importance: Config provides:

- **Continuous configuration recording** capturing resource state changes over time.
- **Compliance checking** through Config rules evaluating whether configurations meet requirements.
- **Relationship tracking** showing dependencies between resources.
- **Configuration history** enabling understanding of how infrastructure evolved and when changes occurred.

CIS alignment: While Config isn’t explicitly mentioned in older CIS versions, it’s essential for implementing many controls. Config enables automated checking of IAM password policies (CIS 1.x), S3 bucket configurations (CIS 2.x), VPC configurations (CIS 4.x), and monitoring for non-compliant resources. Config Conformance Packs provide pre-built CIS Benchmark compliance checking.

Together, CloudTrail and Config provide:

- CloudTrail answers "who did what" tracking API actions.
- Config answers "what changed and when" tracking configuration state.
- CloudTrail enables reactive investigation after incidents.
- Config enables proactive compliance before problems occur.
- CloudTrail provides event-level detail for forensics.
- Config provides configuration snapshots for compliance audits.

Operational implementation:

- Deploy CloudTrail and Config organization-wide using Organizations.
- Centralize logs in dedicated security account preventing tampering by workload account owners.
- Enable automated monitoring and alerting on both services.
- Integrate with SIEM for correlation.
- Use Config Aggregator for multi-account visibility.

Not having these services means operating blind—unable to investigate incidents, prove compliance, or detect configuration drift. They’re foundational to AWS security posture.

How would you address vulnerabilities identified by AWS Inspector that are related to the AWS CIS Benchmark?

AWS Inspector scans EC2 instances, container images, and Lambda functions for software vulnerabilities and network exposures. When Inspector findings relate to CIS Benchmark, addressing them requires systematic approach.

Understanding Inspector findings: Inspector generates findings with:

- Severity (critical, high, medium, low, informational).
- CVE identifiers for software vulnerabilities.
- CIS Benchmark rule IDs when finding relates to specific benchmark recommendation.
- Affected resources.
- Remediation recommendations.

Inspector assesses against CIS benchmarks for operating systems (CIS Amazon Linux Benchmark, CIS Ubuntu Benchmark, etc.) checking OS-level configurations, not AWS service configurations (which Config handles).

Prioritization:

- Critical and high severity findings with known exploits get immediate attention.
- Findings in internet-facing instances prioritized over internal.
- Production environments remediated before development.
- Instances handling sensitive data prioritized.
- Use CVSS scores and exploit availability to risk-rank.

Remediation workflow:

For software vulnerabilities: Inspector identifies outdated packages with CVEs.

- Use Systems Manager Patch Manager to apply updates—create patch baseline including identified CVEs, schedule maintenance window for patching, test patches in non-production first, and apply to production during approved change window.
- For immutable infrastructure, rebuild AMIs with updated packages and redeploy instances.

For CIS Benchmark OS configuration issues: Inspector flags non-compliant OS settings like weak SSH configuration, unnecessary services running, or file permission issues.

- Create Systems Manager State Manager associations applying compliant configurations: Ansible playbooks or shell scripts implementing CIS recommendations, applied automatically to instances with specific tags, and verified through Inspector rescanning.
- Alternatively, update golden AMI build process including CIS hardening scripts ensuring new

instances deploy compliant.

Automated response:

- For low-risk remediations, implement automatic response: EventBridge rule triggers on new Inspector findings, Lambda function evaluates finding type and severity, if finding type is "patchable software vulnerability" and severity is medium/low, Lambda invokes Systems Manager Run Command applying patch, and Inspector rescans verifying remediation.

For findings requiring manual intervention:

- Create incident tickets in ServiceNow/Jira with finding details, severity, affected resource, and remediation steps.
- Assign to instance owner team with SLA based on severity (24 hours for critical, 7 days for high).
- Track remediation progress with automated reminders for SLA violations.
- Verify fix through Inspector rescan.

Prevention:

- Update AMI build pipelines including CIS hardening: Use CIS-compliant base AMIs from AWS Marketplace or build custom AMIs with hardening scripts.
- Implement automated AMI scanning with Inspector before approval.
- Only approve AMIs passing CIS compliance checks.
- Periodically rebuild AMIs with latest patches.
- Implement immutable infrastructure preventing configuration drift—instances replaced not patched, reducing "snowflake" systems.

Continuous monitoring:

- Inspector runs continuous assessments detecting new vulnerabilities.
- EventBridge integration with Security Hub aggregates findings.
- Dashboards track vulnerability trends and mean-time-to-remediate.
- Regular reviews identify systemic issues requiring architectural changes.

Exceptions and risk acceptance:

- Some findings may be false positives or accepted risks—document justification for not remediating.
- Implement compensating controls (WAF protecting vulnerable application).
- Track exceptions with regular re-evaluation.

Example scenario: Inspector finds CIS Ubuntu Benchmark violation - weak SSH configuration allowing root login on production web servers. Remediation:

- Update launch template user data hardening SSH configuration (`PermitRootLogin no`, `PasswordAuthentication no`).

- Create Systems Manager State Manager association applying SSH hardening to existing instances.
- Terminate and re-launch instances from updated template during maintenance window.
- Verify with Inspector showing compliance.

The key is treating Inspector findings as actionable security work items with clear ownership, SLAs, and tracking through remediation.

CloudTrail vs. CloudWatch and explain in-depth from a security perspective.

CloudTrail and CloudWatch serve different but complementary security purposes and are often confused.

CloudTrail - Audit Logging: CloudTrail is AWS's audit logging service recording **every API call** made in your AWS account. It captures who (`userIdentity`), what (`eventName` like `RunInstances` or `PutObject`), when (`eventTime`), where (`sourceIPAddress`), and what the result was (`errorCode` or success). CloudTrail logs are immutable records of account activity providing:

- **Forensic evidence** for investigations.
- **Compliance audit trail** proving who did what.
- **Governance** tracking infrastructure changes.
- **Anomaly detection** identifying unusual API patterns.

CloudTrail is **always retrospective**--it tells you what happened after the fact. From security perspective, CloudTrail is your **primary investigation tool** during incidents, **compliance evidence** for auditors, and **source of truth** for what occurred in your account.

CloudTrail data events track object-level operations in S3, Lambda function executions, and DynamoDB item operations providing granular activity logs.

Security use cases:

- Investigating who deleted S3 bucket.
- Tracing privilege escalation attempt through IAM API calls.
- Proving compliance during audit by showing MFA enforcement.
- Detecting insider threats by analyzing user behavior patterns.
- Identifying compromised credentials by tracking unusual API sources.

CloudWatch - Monitoring and Alerting: CloudWatch is AWS's monitoring service tracking **metrics**, **logs**, and **events** for operational and security visibility. It's designed for **real-time awareness** not historical investigation.

CloudWatch has three main components:

- **Metrics** (numeric data points like CPUUtilization, NetworkIn, custom application metrics) with **alarms** triggering on threshold violations.
- **Logs** aggregating application and system logs with **metric filters** extracting patterns and **Insights** for querying.
- **Events/EventBridge** routing AWS service events to targets for automation.

From security perspective, CloudWatch provides **real-time detection** through alarms and rules, **operational security** monitoring system health and performance, and **custom application security** logs.

Security use cases:

- Alerting when root account is used via metric filter on CloudTrail logs in CloudWatch Logs.
- Detecting failed SSH attempts via metric filter on auth.log.
- Triggering automated response when security group changes occur via EventBridge.
- Monitoring GuardDuty findings and auto-remediating through Lambda.

Key differences from security perspective:

Aspect	CloudTrail	CloudWatch
Nature	"what happened" (audit log)	"what's happening now" (monitoring)
Timeframe	Retrospective investigation tool	Real-time alerting tool
Scope	Logs API actions only	Handles metrics, logs, and events from all sources
Use during incidents	Forensic analysis tracing attacker actions	Detecting attack in progress and alerting
Compliance	Provides audit evidence	Provides operational visibility

Integration for security: The two work together powerfully:

- CloudTrail logs stream to CloudWatch Logs.
- Metric filters on CloudTrail logs extract security events (failed console logins, IAM changes, root account usage).
- CloudWatch Alarms trigger on metric filter matches alerting security team.
- EventBridge rules on CloudTrail API calls invoke Lambda for automated response.

Example workflow: Attacker attempts to disable CloudTrail.

1. CloudTrail logs the `StopLogging` API call.
2. CloudTrail log delivered to CloudWatch Logs within minutes.
3. Metric filter matches `eventName: StopLogging`.

4. CloudWatch Alarm triggers sending SNS notification and invoking Lambda.
5. Lambda re-enables CloudTrail and alerts security team.

Without CloudTrail, no record of the attempt exists. Without CloudWatch, detection requires manual log review hours later instead of real-time alerting.

Best practice: Enable both CloudTrail for comprehensive audit logging across all regions and accounts, and CloudWatch Logs/EventBridge for real-time security monitoring and automated response. They're complementary, not alternative choices.

Why is IMDSv1 vulnerable to SSRF, and can you explain it?

IMDSv1 (Instance Metadata Service version 1) is vulnerable to Server-Side Request Forgery (SSRF) attacks because it uses simple HTTP GET requests without authentication to 169.254.169.254, allowing any code running on the instance to access sensitive metadata including IAM credentials.

How IMDSv1 works: Applications on EC2 instances retrieve metadata by making HTTP requests:

- `curl http://169.254.169.254/latest/meta-data/` returns instance metadata.
- `curl http://169.254.169.254/latest/meta-data/iam/security-credentials/ROLE-NAME` returns temporary IAM credentials (AccessKeyId, SecretAccessKey, SessionToken) for instance's role.

These credentials allow making AWS API calls with the instance role's permissions. IMDSv1 has **no authentication**--any HTTP GET to `169.254.169.254` returns data.

SSRF vulnerability: SSRF occurs when an attacker can make a server-side application perform HTTP requests to arbitrary URLs. Common SSRF vectors include:

- Web applications accepting user-supplied URLs (image proxies, URL fetchers, webhook endpoints).
- XML parsing vulnerabilities (XXE allowing external entity references).
- PDF generators or document converters following URLs.

Attack scenario:

1. Web application running on EC2 has URL parameter: `https://myapp.com/fetch?url=USER_INPUT`.
2. Attacker provides: `https://myapp.com/fetch?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/WebAppRole`.
3. Application server makes HTTP request to IMDS (thinking it's fetching legitimate content).
4. IMDS returns IAM credentials in response.
5. Application returns credentials to attacker in HTTP response.
6. Attacker now has IAM credentials for instance role with full permissions.

Why this works with IMDSv1:

- IMDS is accessible via simple GET requests.
- `169.254.169.254` is always reachable from instance (link-local address).
- No authentication required—any HTTP client on instance can access.
- HTTP redirect chains work (attacker can use redirect to hide IMDS URL).
- No request origin validation.

Real-world example: Capital One breach (2019) involved SSRF vulnerability in web application firewall configuration allowing attacker to query IMDS, retrieve IAM credentials, and access S3 buckets containing customer data.

Why developers might create SSRF vulnerabilities:

- Accepting user input for URLs (fetch image from URL, webhook callbacks).
- Insufficient URL validation allowing internal addresses.
- Following redirects without checking destination.
- XML/XXE vulnerabilities in parsers.

Defense against SSRF in IMDSv1:

- Input validation blocking private IP ranges (`169.254.x.x`, `10.x.x.x`, `192.168.x.x`).
- URL allowlisting permitting only specific domains.
- Disable HTTP redirects or validate redirect destinations.
- Use IMDSv2 which prevents SSRF exploitation.

The fundamental issue is IMDSv1 trusts any HTTP request from the instance—it can't distinguish between legitimate application code and attacker-controlled requests made through SSRF. This makes IMDSv1 dangerous in environments with potential SSRF vulnerabilities.

Have you implemented IMDSv2, and how does it fix SSRF?

Yes, I've implemented IMDSv2 across environments. IMDSv2 (Instance Metadata Service version 2) fixes SSRF vulnerabilities through **session-oriented authentication** requiring additional steps that SSRF attacks typically can't complete.

How IMDSv2 works: Instead of simple GET requests, IMDSv2 requires two-step process:

Step 1 - Get session token: Application makes PUT request with custom header to special endpoint:

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

This returns a session token valid for specified TTL (1-21600 seconds).

Step 2 - Use token for metadata requests: Include token in header for actual metadata requests:

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

Why this defeats SSRF:

- **PUT method requirement**: Most SSRF vulnerabilities only allow GET requests (URL fetchers, image proxies, webhooks typically only support GET). PUT is blocked by many vulnerable applications.
- **Custom headers required**: SSRF through web browsers or simple HTTP clients typically can't set custom headers. The vulnerable application would need to support header injection, which is rarer.
- **Two-step process**: Attacker needs to first retrieve token, then use it in subsequent request. Most SSRF vulnerabilities don't allow chaining multiple requests with token from first response.
- **Token TTL**: Tokens expire, requiring repeated authentication which SSRF exploits typically can't maintain.
- **Hop limit**: IMDSv2 implements IPv4 packet TTL/hop limit of 1, preventing metadata requests that traverse network hops. Docker containers or forwarded requests fail.

Implementation:

Gradual migration:

- Start by enabling IMDSv2 support alongside IMDSv1 (default).
- Update applications to use IMDSv2 API (add token retrieval logic).
- Test thoroughly ensuring applications work.
- Then enforce IMDSv2-only gradually.

Enforcement at instance level: Launch instances with metadata options requiring IMDSv2:

```
aws ec2 run-instances --metadata-options
"HttpTokens=required,HttpPutResponseHopLimit=1"
```

- `HttpTokens=required` enforces IMDSv2 (vs `optional` allowing both).
- `HttpPutResponseHopLimit=1` prevents forwarded requests.

For existing instances:

```
aws ec2 modify-instance-metadata-options --instance-id i-1234567890abcdef0 --http
```

```
-tokens required --http-endpoint enabled
```

Launch template updates: Modify launch templates and Auto Scaling groups to use IMDSv2:

```
"MetadataOptions": {
  "HttpTokens": "required",
  "HttpPutResponseHopLimit": 1,
  "HttpEndpoint": "enabled"
}
```

Application code updates:

- Update SDKs (AWS SDKs automatically support IMDSv2).
- For custom HTTP clients, implement token retrieval and usage.

Organizational enforcement: Use SCPs preventing instance launch without IMDSv2:

```
{
  "Effect": "Deny",
  "Action": "ec2:RunInstances",
  "Resource": "arn:aws:ec2:*:*:instance/*",
  "Condition": {
    "StringNotEquals": {
      "ec2:MetadataHttpTokens": "required"
    }
  }
}
```

Use Config rules detecting instances not requiring IMDSv2 and auto-remediate or alert.

Monitoring:

- Track IMDSv2 adoption using Config rules.
- Detect IMDSv1 usage through CloudWatch metrics.
- Transition timeline with target dates for enforcement.

Benefits beyond SSRF:

- Defense in depth even if SSRF exists.
- Forced authentication for metadata access.
- Hop limit prevents container escape scenarios.
- Aligns with AWS security best practices.

IMDSv2 should be standard for all new instances, with migration plan for existing workloads. It fundamentally changes IMDS from unauthenticated to session-authenticated, making exploitation through SSRF extremely difficult.

What is the Instance Metadata Service (IMDS `169.254.169.254`) in AWS, and why is it a potential security concern for EC2 instances? Explain how attackers can abuse the IMDS to compromise an EC2 instance's security.

IMDS is a service available to all EC2 instances at the link-local IP `169.254.169.254` providing instance metadata including instance ID, AMI ID, network configuration, IAM role credentials, user data, and security groups. It's intended for instances to discover information about themselves and retrieve temporary IAM credentials for API calls.

Why it's a security concern: IMDS exposes **IAM credentials** at `/latest/meta-data/iam/security-credentials/ROLE-NAME` which are temporary but fully functional AWS credentials with all permissions granted to the instance role. If attackers can query IMDS (through SSRF or other means), they obtain these credentials and can make AWS API calls as the instance. This enables:

- **Privilege escalation** if instance role has broad permissions.
- **Lateral movement** accessing other AWS resources the role can reach.
- **Data exfiltration** downloading S3 buckets, querying databases, or accessing secrets.
- **Persistence** creating backdoor access or additional credentials.

Attack vectors:

- **SSRF (Server-Side Request Forgery)**: Most common—attacker exploits application vulnerability making server query IMDS on their behalf (covered in questions 84-85).
- **Application vulnerabilities**: Command injection in application allows attacker to run `curl 169.254.169.254/...`, local file inclusion reading credentials from application's environment which retrieved them from IMDS, or XXE in XML parsers fetching IMDS URLs.
- **Container escape**: If attacker escapes container to host, they can query IMDS getting host instance credentials.
- **Compromised application code**: Malicious dependencies, supply chain attacks, or backdoored code querying IMDS and exfiltrating credentials.
- **User data script execution**: If attacker can modify user data (through separate vulnerability), script runs with IMDS access.

Attack chain example:

1. Attacker finds SSRF in web application.

2. Uses SSRF to query `http://169.254.169.254/latest/meta-data/iam/security-credentials/`, gets role name "WebServer-Role".
3. Queries full credentials at that endpoint, receives AccessKeyId, SecretAccessKey, SessionToken.
4. Uses credentials to call `aws s3 ls` discovering accessible S3 buckets.
5. Downloads sensitive data from S3.
6. Queries `aws iam get-user` or similar discovering what permissions role has.
7. If role has `iam:CreateAccessKey` or similar, creates persistent access.

What makes this particularly dangerous:

- Credentials are **dynamically rotated** but valid for hours (default 6 hours).
- Attack is **invisible** to instance—no unusual process or network activity.
- Credentials work from anywhere (not just the instance).
- Many instance roles have **excessive permissions** violating least privilege.

Real-world impact: Capital One breach used SSRF to access IMDS obtaining credentials for overprivileged role accessing customer data S3 buckets. Multiple vulnerabilities in popular software (Apache Struts, etc.) enabled IMDS access. Cloud metadata services across providers (not just AWS) have been exploit targets.

Mitigations:

- Use IMDSv2 requiring authentication.
- Implement least privilege on instance roles.
- Network segmentation limiting what compromised instances can access.
- SSRF prevention in applications.
- Monitoring for unusual IMDS access patterns.
- Avoid storing sensitive data accessible to instance roles.

IMDS is powerful operational tool but security liability if not properly protected, making IMDSv2 enforcement and least privilege role design critical.

How can organizations protect against unauthorized access to IAM credentials via the IMDS, and what best practices should be followed to mitigate this risk?

IAM credentials in IMDS create significant security surface.

Security implications:

- Credentials are **fully functional AWS API credentials** with all permissions of the instance role.
- They're **accessible to any process** running on instance including compromised applications or malware.
- Credentials are **retrievable via SSRF** as discussed earlier.
- They're **long-lived enough** (hours) for substantial damage.
- Credentials work **from any location** not just the instance enabling exfiltration.
- Many roles have **excessive permissions** amplifying impact.

The core issue is that IMDS credentials blur the security boundary—application compromise effectively becomes AWS account compromise if role is over-permissioned.

Protection strategies:

IMDSv2 enforcement:

- Require IMDSv2 on all instances preventing SSRF-based credential theft.
- Use SCPs and Config rules ensuring compliance.
- Update applications to support IMDSv2.

Least privilege IAM roles: Most critical mitigation—instance roles should have minimum permissions required:

- Avoid wildcard permissions (`s3:*`, `Resource: "*"`).
- Use specific resource ARNs in policies.
- Implement condition statements restricting when/how permissions can be used.
- Regularly review and remove unused permissions with Access Analyzer.

Role session duration limits: Reduce maximum session duration for instance roles from default 12 hours to minimum needed (1 hour if feasible), requiring more frequent credential rotation limiting window for stolen credentials.

Network security:

- Implement security groups allowing only necessary outbound connections.
- Use VPC endpoints for AWS services preventing credential use from external networks (some attacks).
- Deploy instances in private subnets.
- Use network segmentation limiting lateral movement.

Application security:

- Prevent SSRF vulnerabilities through input validation, URL allowlisting, and disabling redirects.
- Implement WAF protecting against common injection vulnerabilities.

- Regular application security testing including SSRF checks.
- Dependency scanning preventing vulnerable libraries.

Monitoring and detection:

- CloudTrail logging all API calls made with instance credentials.
- Alerts on unusual API activity from instance roles (calls from unexpected regions, services not normally used, bulk operations).
- GuardDuty detecting credential compromise indicators.
- Behavioral analysis establishing baselines for each role's normal activity.

Credential scoping:

- Use different IAM roles for different workloads on same instance when possible (multiple containers with different task roles in ECS).
- Avoid shared instance roles across unrelated applications.
- Implement tagging and monitoring per role understanding exposure.

Disable IMDS when not needed: For instances not requiring AWS API access, disable IMDS entirely: `--metadata-options "HttpEndpoint=disabled"`. For containerized workloads, use task roles (ECS) or service accounts (EKS) instead of instance roles providing container-level credential isolation.

Secrets management: Don't rely solely on instance role credentials for application secrets, use Secrets Manager or Parameter Store for sensitive values with separate access controls, and implement application-level authentication to AWS services when possible.

Incident response: Automated response to suspected credential compromise: revoke credentials by modifying role trust policy temporarily, isolate affected instance via security group changes, snapshot for forensics, and rotate affected credentials.

Testing:

- Regularly test SSRF vulnerabilities in applications.
- Attempt to access IMDS from containers verifying isolation.
- Red team exercises simulating credential theft.

Organizational policies:

- Require security review for all new IAM roles.
- Automated scanning for overly permissive roles.
- Quarterly access reviews removing unused permissions.
- Training developers on IMDS security risks.

Example secure configuration: Instance role for web server only allows:

- Read from specific S3 bucket for static assets.
- Write to CloudWatch Logs for logging.
- Read from Secrets Manager for database credentials—nothing else.

Even if credentials stolen, attacker can't access other S3 buckets, modify IAM, or launch resources. Combine this with IMDSv2, short session duration, and SSRF prevention creating defense in depth.

The key is treating instance role credentials as highly sensitive despite being temporary, implementing multiple protective layers rather than relying on single control.

When should you use TGW (Transit Gateway), and is there any security improvement for using this?

Transit Gateway should be used when you need to interconnect many VPCs, VPN connections, or Direct Connect gateways at scale, and it provides several security benefits.

When to use TGW:

Many VPC connections - with 10+ VPCs, full mesh peering becomes unmanageable (45 peering connections for 10 VPCs, 4,950 for 100). TGW provides hub-and-spoke reducing to n connections.

Centralized routing control - when you need consistent routing policies across environments, TGW route tables provide central control point.

Network segmentation at scale - isolating production from development, different business units, or multi-tenant environments while allowing selective connectivity.

Hybrid cloud - connecting on-premises networks to multiple VPCs through single VPN or Direct Connect attachment instead of per-VPC connections.

Inspection architecture - routing traffic through centralized security VPC for firewall inspection, IDS/IPS, or DLP.

Multi-region connectivity - TGW peering connects regions with private networking.

Security improvements:

Centralized traffic inspection - route all inter-VPC traffic through security VPC attachment with third-party firewalls, network intrusion detection systems, or DLP appliances. This is impractical with mesh peering. Traffic flows VPC A → TGW → Security VPC (inspection) → TGW → VPC B. Configure route tables directing traffic through inspection VPC before destination.

Simplified network segmentation - create isolated routing domains with TGW route tables:

- Production route table (prod VPCs can communicate).

- Development route table (dev VPCs isolated from prod).
- Shared services route table (accessible by both).

Association and propagation controls prevent unauthorized connectivity.

Reduced attack surface - fewer network paths to secure compared to mesh peering, centralized chokepoint for monitoring and control, and simplified security group management (connection to TGW instead of many peers).

Consistent security policies - apply uniform network policies across environment, centralized logging of network flows through TGW, and standardized connectivity patterns across teams/applications.

Hybrid connectivity security - single VPN or Direct Connect attachment to TGW serves all VPCs versus per-VPC connections, centralized control of what on-premises can access, and dedicated route tables for hybrid connectivity isolating from VPC-to-VPC traffic.

Network monitoring and visibility - VPC Flow Logs from TGW attachments showing inter-VPC traffic, CloudWatch metrics on TGW providing network visibility, and traffic trending and anomaly detection from centralized viewpoint.

Compliance benefits - clear network segmentation for regulatory requirements (PCI DSS, HIPAA), audit trail of network connectivity through TGW configuration history in CloudTrail, and documented network architecture for compliance assessments.

Example secure architecture:

- Three TGW route tables:
  a. Production table—prod VPCs can communicate with each other and shared services, explicitly deny routes to dev.
  b. Development table—dev VPCs communicate internally only.
  c. Shared Services table—routes to both prod and dev for centralized services (Active Directory, monitoring).
- Inspection VPC attachment forces traffic through NGFWs before reaching destination.
- On-premises attachment only has routes to DMZ VPC, not internal workloads.

This creates defense in depth with multiple security controls, prevents unauthorized access paths, and provides centralized visibility—all difficult or impossible with VPC peering alone.

Considerations:

- TGW costs more than VPC peering (hourly attachment fee plus data processing).
- Adds single point of failure (mitigated by TGW high availability across AZs).
- Requires careful route table design preventing unintended connectivity.

For small deployments (< 5 VPCs) with simple connectivity, peering may be sufficient. For large, complex environments requiring strong segmentation and inspection, TGW provides superior security architecture.

Why is a security group named "default" with ports 22, 25, 53, 80, 443, 8080, 6443, 3679, 3306, 9001 open an issue?

This is extremely dangerous for multiple reasons.

Overly permissive access: Opening numerous ports creates massive attack surface. Each port is potential entry point for attackers. Having all these simultaneously is almost never necessary and violates least privilege.

Sensitive ports exposed:

- Port 22 (SSH) - administrative access, should be restricted to management networks or bastion hosts, never 0.0.0.0/0.
- Port 3306 (MySQL) - database access should never be internet-facing, only accessible from application tier.
- Port 9001 - various uses including AWS Lambda runtime API (discussed in question 60), potential SSRF target.
- Port 6443 - Kubernetes API server, extremely sensitive administrative interface.
- Port 25 (SMTP) - email, often abused for spam, rarely needed.

"Default" security group: The default security group is automatically assigned to resources if no security group is specified. Developers often launch instances without explicitly choosing security group, defaulting to this. If default is permissive, unintentional exposure is widespread. Every forgotten security group specification becomes a vulnerability.

Source IP assumption: The question doesn't specify source, but if these ports allow 0.0.0.0/0 (internet), it's catastrophic. Even if limited to VPC CIDR, it's overly broad unless specific application requires it.

Common attack scenarios:

- Automated scanners find port 22 or 3306 open.
- Brute force attacks against SSH or database.
- Exploitation of unpatched services on these ports.
- Port 3306 open enables database exploitation and data theft.
- Kubernetes API (6443) exposed enables cluster takeover.

Multiple services implication: No single instance should need all these ports, suggesting either monolithic architecture (anti-pattern) or copy-paste security group reuse without thought.

Best practices violated:

- Default-deny approach—start with no access, add only necessary ports.
- Service-specific security groups—web servers have different security groups than databases.
- Application-tier security groups reference each other instead of opening ports to 0.0.0.0/0.
- Administrative access (SSH) only from specific management security group or through Session Manager.

Remediation:

- Audit default security group immediately identifying attached resources.
- Create purpose-specific security groups (web-tier-sg, app-tier-sg, db-tier-sg) with appropriate ports.
- Migrate resources to specific security groups.
- Lock down default security group to deny all or minimal access.
- Implement Config rule detecting usage of default security group alerting for violations.
- Use SCPs preventing default security group usage in production accounts.
- Educate teams on security group best practices.

Proper design:

- Web tier security group: 443 from 0.0.0.0/0, 22 from management-sg.
- App tier security group: 8080 from web-tier-sg, 22 from management-sg.
- Database tier security group: 3306 from app-tier-sg only, 22 from management-sg or use Session Manager (no SSH).

This micro-segmentation prevents lateral movement and limits blast radius. A permissive default security group with many ports open is security anti-pattern indicating lack of network security understanding and creating significant vulnerability.

Can you explain how to use and when to use Access Key ID and Principal ID with one example?

Access Key ID and **Principal ID** serve different purposes in AWS IAM.

Access Key ID: This is the public identifier for IAM user long-lived credentials or temporary STS credentials.

- Format: `AKIA...` for long-lived IAM user keys, `ASIA...` for temporary STS credentials.
- Access Key ID is used with Secret Access Key for AWS API authentication via AWS SDK or CLI.

When to use:

- Programmatic access from external systems (on-premises applications, CI/CD systems like Jenkins).
- Third-party integrations requiring AWS access.
- Development/testing with AWS CLI.

Best practices:

- Prefer IAM roles over access keys wherever possible (eliminate long-lived credentials).
- Rotate access keys regularly (90 days).
- Never commit access keys to code repositories.
- Use temporary credentials (STS) instead of IAM user keys.
- Monitor access key usage with credential reports.

Principal ID: This is a unique identifier for an IAM principal (user, role, or federated user) that persists even if the principal name changes.

- Format: `AIDA...` for IAM users, `AROA...` for IAM roles, `AGPA...` for IAM groups.
- Principal ID never changes even if you rename the user/role, making it reliable for tracking entities across name changes.

When to use:

- Resource-based policies where you need consistent principal reference regardless of renames.
- CloudTrail log analysis tracking specific entity's actions even after renames.
- Audit trails and compliance where principal identity must be definitive.
- Detecting anomalous behavior by specific principal.

Example scenario - Access Key ID:

1. Application running on-premises needs to upload files to S3 bucket `data-uploads`.
2. Create IAM user `OnPremUploader` with programmatic access generating Access Key ID and Secret Access Key.
3. Grant minimal S3 permissions: `{"Effect": "Allow", "Action": ["s3:PutObject"], "Resource": "arn:aws:s3:::data-uploads/*"}`.
4. Application uses Access Key ID and Secret in API calls: `aws s3 cp file.txt s3://data-uploads/ --profile onprem`.
5. CloudTrail logs show Access Key ID `AKIAIOSFODNN7EXAMPLE` made PutObject call.
6. If this key is compromised, you rotate it generating new Access Key ID.

Example scenario - Principal ID:

1. You create IAM role `DataScientist-Role` with Principal ID `AROAI23HX7MHQEXAMPLE`.

2. Grant S3 bucket policy allowing this role: `{"Principal": {"AWS": "arn:aws:iam::ACCOUNT:role/DataScientist-Role"}}`.
3. CloudTrail logs show activity from `principalId: AROAI23HX7MHQEXAMPLE`.
4. Later, you rename role to `DataAnalyst-Role` updating ARN. Principal ID remains `AROAI23HX7MHQEXAMPLE`.
5. Historical CloudTrail logs are still valid—you can query all activity by this principal across name change using consistent Principal ID.
6. Bucket policy breaks after rename (ARN changed), but you can update policy.
7. If using Principal ID directly (less common): `{"Principal": {"AWS": "AROAI23HX7MHQEXAMPLE"}}`, policy survives rename.

Another example - Anomaly detection: Security team analyzing CloudTrail finds unusual API calls. Filter by `userIdentity.principalId: AIDAI23HX7ABCEXAMPLE` shows all actions by specific IAM user even if user was renamed during investigation timeframe. Access Key ID might change (rotation), but Principal ID is constant.

Key differences:

- Access Key ID is credential identifier for authentication, Principal ID is entity identifier for authorization and auditing.
- Access Key ID changes when rotated, Principal ID never changes.
- Access Key ID used in API calls, Principal ID used in logs and policies.

In practice, you'll use Access Key ID for configuring authentication (providing credentials to applications) and Principal ID for auditing and security investigations (tracking who did what). Modern best practice is minimizing Access Key ID usage entirely, preferring IAM roles with temporary credentials, while Principal ID remains important for audit and tracking purposes.

Explain the given IAM policy and its purpose.

NOTE Since no specific policy was provided in your question, I'll explain how I'd approach answering this in an interview with a sample policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": "arn:aws:s3:::company-data-${aws:username}/*",
      "Condition": {
        "IpAddress": {
          "aws:SourceIp": "203.0.113.0/24"
        }
      }
    }
  ]
}
```

My analysis: This is an IAM policy granting S3 object access with specific restrictions.

Purpose: Allow users to read and write objects in their personal S3 bucket folder while enforcing network-based access control.

Breakdown:

- **Effect: Allow** - This is a permissive policy granting access.
- **Actions**: `s3:GetObject` allows downloading/reading objects, `s3:PutObject` allows uploading/writing objects. Notably missing: `s3:DeleteObject` (users can’t delete), `s3:ListBucket` (users can’t list bucket contents), and `s3:GetObjectVersion` (no version access).
- **Resource**: `arn:aws:s3:::company-data-${aws:username}/ uses policy variable ${aws:username} which resolves to the IAM user□s name. This creates user-specific paths—user "alice" can access company-data-alice/`, user "bob" accesses `company-data-bob/. This prevents users from accessing each other□s data. The /` applies to objects, not the bucket itself.
- **Condition**: `IpAddress` condition with `aws:SourceIp: 203.0.113.0/24` restricts access to specific IP range, likely corporate network. Users must be on corporate network to access S3, preventing access from home or public networks.

Use case: This policy is designed for a scenario where employees need personal S3 storage accessible only from office network—perhaps for work-related file storage with data residency controls.

Security considerations:

- GOOD—least privilege (only necessary actions), user isolation through path variables, network-based access control, and no delete permissions preventing accidental data loss.
- CONCERNS—IP-based security can be bypassed via VPN or compromised corporate machines, lacks MFA requirement for sensitive operations, no encryption requirement (should add `s3:x-amz-server-side-encryption`), and missing `s3:ListBucket` might impact usability.

Recommendations:

- Add MFA condition for PutObject to prevent unauthorized uploads.
- Require encryption: `"StringEquals": {"s3:x-amz-server-side-encryption": "AES256"}`.
- Add `s3:ListBucket` with resource condition limiting to user’s prefix.

- Implement VPC endpoint condition instead of IP for stronger network control.
- Consider time-based conditions limiting access to business hours.

Explain the given policy and identify any issues with it.

NOTE Again, since no specific policy was provided, I'll demonstrate with a problematic policy example:

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

Analysis: This is a wildcard administrative policy granting unrestricted access to all AWS services and resources.

Critical issues identified:

Issue 1 - Violates least privilege: `Action: "*"` grants every AWS action including dangerous operations like `iam:CreateUser`, `iam:AttachUserPolicy`, `s3:DeleteBucket`, `ec2:TerminateInstances`, `organizations:LeaveOrganization`. This is administrative access that should be extremely restricted.

Issue 2 - No resource restrictions: `Resource: "*"` applies permissions to every resource in the account across all services and regions. User can modify any resource regardless of ownership or environment (production vs. development).

Issue 3 - No conditions: Complete absence of condition statements means no MFA requirement, no IP restrictions, no time-based access, and no service-specific limitations. Access works from anywhere, anytime, by anyone with these credentials.

Issue 4 - Privilege escalation risk: With `iam:*` permissions, users can grant themselves additional permissions, create new admin users, attach policies to other users, or modify their own policies maintaining persistent access.

Issue 5 - Blast radius: If credentials with this policy are compromised, attacker has full account control enabling complete data exfiltration, resource deletion/modification, billing fraud through resource creation, and persistent access through backdoor accounts/roles.

Issue 6 - Compliance violations: Most compliance frameworks (PCI DSS, HIPAA, SOC 2) prohibit wildcard permissions. Auditors will flag this as high-severity finding.

Issue 7 - Lack of accountability: No way to justify why any specific action is needed when everything is allowed. Can't implement separation of duties or least privilege.

Recommendations:

- Replace with role-based access granting only necessary permissions for specific job functions.
- Implement permission boundaries limiting maximum permissions users can grant.
- Require MFA for administrative actions: `"Condition": {"Bool": {"aws:MultiFactorAuthPresent": "true"}}`.
- Use time-based conditions for temporary elevated access.
- Implement SCPs preventing certain dangerous actions organization-wide.
- Enable comprehensive CloudTrail logging and alerting on administrative actions.
- Conduct regular access reviews identifying unused permissions.

Better approach: Create specific policies for different roles—developers get EC2/S3/Lambda permissions in dev account, operations gets read-only production access plus specific deployment permissions, and administrators get elevated but scoped permissions with MFA requirement and audit trails.

This policy represents worst-case IAM configuration and should never be used except potentially for break-glass emergency access with extreme monitoring and time-limited access.

What comes to your mind when a service needs cross-account access?

Cross-account access immediately triggers several security considerations.

First thought - Why?: Understand the business justification—is this third-party vendor access, multi-account architecture (separate prod/dev/security accounts), acquisition/merger requiring inter-organization access, or centralized service (logging, backup) needing data from workload accounts. The purpose drives security controls.

IAM role assumption: Preferred method over IAM users. Create role in target account (where resources live) with trust policy allowing source account principals to assume it. This provides:

- Temporary credentials, better than long-lived access keys.
- Enables audit trail of who assumed role when.
- Allows centralized permission management in target account.

Trust boundaries: Cross-account access creates trust relationship requiring careful validation. Trust policy must:

- Specify exact account ID never wildcard.
- Consider requiring External ID preventing confused deputy attacks.
- Implement condition statements (MFA, source IP, time-based).
- Regularly review trust relationships removing unnecessary access.

Least privilege: Grant minimum permissions needed in cross-account role, use resource-level permissions restricting which specific resources can be accessed, implement permission boundaries, and require justification for each permission.

Monitoring and alerting:

- Enable CloudTrail in both accounts logging AssumeRole operations.
- Alert on cross-account assumptions especially from unexpected principals or locations.
- Track resource access from external accounts.
- Implement anomaly detection for unusual cross-account activity.

Resource-based policies: For services supporting them (S3, KMS, SNS, SQS), use resource-based policies as additional authorization layer requiring both IAM permission AND resource policy permission for access—defense in depth.

Security implications: Cross-account access weakens security boundaries—accounts aren't fully isolated, increases attack surface, creates potential for privilege escalation if misconfigured, and complicates audit and compliance.

Network considerations: If cross-account includes network connectivity (VPC peering, Transit Gateway), ensure network segmentation, monitor cross-account traffic with Flow Logs, and implement security groups restricting communication.

Alternatives to consider: Is cross-account access actually necessary or could:

- Data replication work (copy data to requesting account).
- Service integration handle it (cross-account CloudWatch Logs subscription).
- Organizational consolidation eliminate the need.

Risk assessment: Evaluate sensitivity of accessed resources, potential impact if cross-account access compromised, compliance implications, and whether benefits justify risks.

Cross-account access is sometimes necessary but should be exception with strong justification, not default architecture pattern.

What security needs to be taken care of when giving cross-account access & what is confused deputy in IAM?

Security requirements for cross-account access:

Explicit trust policy: Target account role must have trust policy specifying exact source account:

```
"Principal": {"AWS": "arn:aws:iam::SOURCE-ACCOUNT-ID:root"}
```

or better, specific role/user ARN from source account:

```
"Principal": {"AWS": "arn:aws:iam::SOURCE-ACCOUNT-ID:role/CrossAccountRole"}
```

Never use wildcard principals.

External ID for third-party access: When granting access to third-party (vendor, partner), require External ID preventing confused deputy attack:

```
"Condition": {
  "StringEquals": {
    "sts:ExternalId": "unique-secret-value"
  }
}
```

Share External ID securely with third party—they must provide it when assuming role.

Least privilege permissions: Role in target account grants minimum required permissions, use resource-level restrictions, implement condition statements, and avoid wildcard actions/resources.

Permission boundaries: Apply permission boundaries to cross-account roles limiting maximum permissions even if role policy is broadened later.

MFA requirement: For sensitive cross-account access, require MFA:

```
"Condition": {
  "Bool": {
    "aws:MultiFactorAuthPresent": "true"
  }
}
```

preventing compromise of stolen credentials without second factor.

Session duration limits: Reduce maximum session duration for cross-account roles from 12 hours to minimum needed (1-4 hours), requiring frequent re-assumption with fresh authentication.

Monitoring:

- CloudTrail logging in both accounts capturing AssumeRole and subsequent resource access.
- Alerts on new cross-account trust relationships.
- Anomaly detection on cross-account access patterns.
- Regular reviews of cross-account permissions.

Resource-based policies: Use S3 bucket policies, KMS key policies as second authorization layer, require both IAM permission AND resource permission, and prevent policy modification from source account.

Network controls: If cross-account includes network access, implement VPC peering or PrivateLink with proper security groups and use VPC endpoint policies restricting access.

Documentation: Maintain inventory of all cross-account relationships with business justification, owner contacts, and review schedule.

Confused Deputy Problem: This is a security issue where an attacker tricks a more privileged service into performing actions on the attacker's behalf.

How it works:

1. Legitimate service (Deputy) has permissions to access resources.
2. Attacker finds way to invoke Deputy with attacker-controlled parameters.
3. Deputy uses its credentials to access resources.
4. Attacker gains access to resources they shouldn't have.

AWS scenario example:

1. Company A uses Vendor's SaaS service.
2. Vendor's service needs to access Company A's S3 bucket.
3. Company A creates IAM role trusting Vendor's AWS account.
4. Company B (attacker) signs up for same Vendor service.
5. Attacker provides Company A's AWS account ID to Vendor during setup.
6. Vendor's service assumes Company A's role using credentials intended for Company B.
7. Vendor accesses Company A's S3 bucket on behalf of attacker.
8. Company A's bucket is compromised.

Why this happens:

- Trust policy trusts Vendor's account but can't distinguish between Vendor acting for Company A versus Company B.
- Vendor's service uses same AWS account for all customers.

- No way to verify request is for legitimate customer.

External ID solution: External ID solves this.

1. Company A generates unique secret External ID: `"unique-company-a-12345"`.
2. Adds to trust policy: `"Condition": {"StringEquals": {"sts:ExternalId": "unique-company-a-12345"}}`.
3. Shares External ID securely with Vendor (out of band, not through application).
4. Vendor must provide External ID when assuming role.

Now when attacker tries using Company A's account ID, Vendor's service attempts assumption without External ID (attacker doesn't know it) or with wrong External ID, AssumeRole fails due to condition not met, and Company A 's resources remain protected.

Best practices for External ID:

- Generate cryptographically random External ID (not guessable).
- Keep External ID secret between customer and vendor.
- Rotate External ID periodically.
- Document External ID value and purpose.
- Verify vendor implementation actually uses External ID correctly.

Confused deputy protection is critical for any cross-account access involving third parties or multi-tenant services. Always use External ID for third-party vendor access.

Do you agree that we need to enable data encryption at rest by default?

Absolutely yes - encryption at rest should be enabled by default for multiple compelling reasons.

Data protection: Encryption protects against unauthorized physical access to storage media if disks stolen, decommissioned drives not properly wiped, or snapshots accidentally made public. It's defense in depth—even if other controls fail, data remains protected.

Compliance requirements: Most regulatory frameworks mandate encryption at rest:

- PCI DSS requires encryption of cardholder data.
- HIPAA requires PHI encryption.
- GDPR encourages encryption as security measure.
- SOC 2 requires encryption controls.

Enabling by default ensures compliance.

Minimal performance impact: Modern encryption (AES-256) has negligible performance impact with hardware acceleration, AWS services handle encryption transparently, and the cost difference between encrypted vs. unencrypted storage is often zero or minimal. There's no good reason NOT to encrypt.

Prevents accidental exposure:

- Developers forget security configurations.
- Default-encrypted means resources start secure rather than requiring explicit enablement.
- Reduces risk from misconfigurations.
- Simplifies security reviews—encryption is assumed, not checked.

Implementation approaches:

Service-level defaults: Enable default encryption in AWS service settings:

- S3 bucket default encryption at account level.
- EBS encryption by default in EC2 settings.
- RDS automatic encryption for new databases.
- DynamoDB encryption enabled by default.

Account-level enforcement: Use SCPs denying creation of unencrypted resources:

```
{
  "Effect": "Deny",
  "Action": ["s3:PutObject"],
  "Resource": "*",
  "Condition": {
    "StringNotEquals": {
      "s3:x-amz-server-side-encryption": ["AES256", "aws:kms"]
    }
  }
}
```

preventing unencrypted S3 uploads. Similar SCPs for EC2 volumes, RDS instances, etc.

IaC templates:

- Security defaults in CloudFormation/Terraform templates with encryption enabled.
- Policy-as-code (Sentinel/OPA) blocking unencrypted resources in CI/CD.
- Pre-approved modules with encryption baked in.

Monitoring:

- AWS Config rules detecting unencrypted resources.
- Security Hub compliance checks.

- Automated remediation enabling encryption on non-compliant resources.
- Alerts on encryption disabled.

Exceptions: Very few legitimate cases for unencrypted data—perhaps temporary scratch space or public datasets intentionally shared. Even these should be exceptions requiring security review and documentation, not defaults.

Key management considerations: Encryption by default requires thoughtful key management—use AWS-managed keys for simplicity and automatic rotation, customer-managed keys for compliance or key policy control, separate keys per environment/application for isolation, and enable key rotation.

Additional benefits:

- Encryption at rest is often prerequisite for other security features—S3 Object Lock requires versioning, often used with encryption.
- Certain compliance certifications require end-to-end encryption.
- Demonstrates security-first organizational culture to customers and auditors.

Challenges addressed:

- "But encryption impacts performance"--negligible with modern hardware.
- "It’s too expensive"--cost difference is minimal or zero.
- "We don’t store sensitive data"--data classification changes, better safe by default.
- "It’s too complex"--AWS makes it transparent.

None of these justify unencrypted storage.

My position: Encryption at rest should be non-negotiable default enforced through technical controls (SCPs, Config rules) and organizational policy. Exceptions require security committee approval with documented risk acceptance and compensating controls. The question isn’t "why encrypt?" but "why would we ever NOT encrypt?"

What checks do you perform in IAM to ensure a Lambda function triggered by an event works correctly?

Ensuring Lambda function has correct IAM configuration for event-driven execution requires checking multiple components.

Execution role permissions: Lambda function has execution role attached defining what it can do.

- Verify role has necessary permissions for function’s actions—if writing to S3, role needs `s3:PutObject` on specific bucket; if reading from DynamoDB, needs `dynamodb:GetItem`; if calling other AWS services, appropriate permissions.

- Check principle of least privilege—role shouldn't have permissions beyond requirements.
- Validate resource-level permissions are specific ARNs, not wildcards.

Event source permissions: Event source must have permission to invoke Lambda function.

For **S3 trigger**:

- S3 bucket notification configuration must specify Lambda function ARN.
- Lambda resource-based policy must allow S3 to invoke:

  ```
  aws lambda add-permission --function-name MyFunction --statement-id s3-invoke
  --action lambda:InvokeFunction --principal s3.amazonaws.com --source-arn
  arn:aws:s3:::bucket-name --source-account ACCOUNT-ID
  ```

- Verify this permission exists with `aws lambda get-policy`.

For **DynamoDB Streams**:

- Lambda execution role needs `dynamodb:GetRecords`, `dynamodb:GetShardIterator`, `dynamodb:DescribeStream`, `dynamodb:ListStreams` on the stream ARN.
- Event source mapping created with `aws lambda create-event-source-mapping` links stream to function.

For **SNS/SQS**:

- Lambda resource-based policy allows SNS/SQS to invoke function.
- Execution role may need permissions to access message content if encrypted.

For **EventBridge/CloudWatch Events**:

- Rule has Lambda as target.
- Lambda resource-based policy allows `events.amazonaws.com` to invoke.

For **API Gateway**:

- API Gateway has Lambda integration.
- Lambda resource-based policy allows API Gateway to invoke with source ARN specifying API ID.

Trust policy verification: Lambda execution role's trust policy allows Lambda service to assume it: `{"Principal": {"Service": "lambda.amazonaws.com"}}`. Without this, Lambda can't assume role regardless of permissions.

CloudWatch Logs permissions: Lambda needs `logs:CreateLogGroup`, `logs:CreateLogStream`, `logs:PutLogEvents` for logging. AWS creates managed policy `AWSLambdaBasicExecutionRole` with these—verify it's attached or equivalent permissions exist. Without logging permissions, function executes but produces no logs complicating troubleshooting.

VPC access (if applicable): If Lambda in VPC, execution role needs `ec2:CreateNetworkInterface`, `ec2:DescribeNetworkInterfaces`, `ec2:DeleteNetworkInterface` to manage ENIs. Verify through managed policy `AWSLambdaVPCAccessExecutionRole`.

Cross-account permissions: If Lambda accesses resources in different account, verify cross-account role trust relationships and resource-based policies allow access.

Encryption permissions: If function accesses encrypted data (S3 with KMS, encrypted environment variables), execution role needs `kms:Decrypt` on relevant KMS keys. KMS key policy must allow the role to use key.

Testing methodology:

- Use `aws lambda invoke` with test event verifying function executes successfully.
- Check CloudWatch Logs for execution logs and errors.
- Use IAM Policy Simulator testing whether execution role has required permissions:

  ```
  aws iam simulate-principal-policy --policy-source-arn ROLE-ARN --action-names
  s3:PutObject --resource-arns arn:aws:s3:::bucket/key
  ```

- Test event source integration (upload to S3, send SNS message) confirming trigger works.
- Monitor Lambda metrics (Invocations, Errors, Duration) in CloudWatch.

Common issues:

- Missing resource-based policy preventing event source invocation.
- Execution role lacking permissions for function’s AWS SDK calls.
- Trust policy not allowing Lambda service.
- Missing VPC permissions causing function timeout.
- KMS permissions missing preventing decryption.
- Throttling due to insufficient concurrency limits.

Automation:

- Infrastructure as code defining execution role, resource-based policies, and event source configurations together.
- Config rules checking Lambda functions have proper logging permissions.
- Security Hub detecting overly permissive Lambda roles.
- Integration tests in CI/CD validating end-to-end event flow.

Proper IAM configuration for event-driven Lambda requires coordinating execution role (what function can do), resource-based policy (who can invoke function), and event source permissions (how trigger connects). Testing all three ensures reliable event processing.