**AWS Backlog Clarifications**

**Personal Learning Journey**

**Personal Learning Summary**

This document outlines key topics discussed during the recent review meeting that required further clarification or deeper understanding from my side. It captures unresolved items from the AWS backlog, including specific service configurations, security event simulations, and operational best practices. The goal is to ensure alignment with team expectations and to proactively address knowledge gaps for improved performance and contribution.

**Complete Learning Journey**

1. **CIDR IP Range Understanding** - Mathematical foundation and enterprise network planning
2. **Public vs Private Subnets** - Security architecture and defense in depth
3. **MFA + CLI Security** - Enterprise authentication patterns and temporary credentials
4. **Two MFA Codes Requirement** - TOTP synchronization and security reasoning
5. **Breakglass Users** - Emergency access strategy and business continuity
6. **PEM File Security** - SSH key management and stealth restriction techniques
7. **Security Groups vs NACLs** - Port security, threat detection, and audit procedures
8. **Network Interface Concepts** - Virtual networking, subnet relationships, capacity planning
9. **Route Table Architecture** - Network segmentation and enterprise best practices
10. **HTTP Access Troubleshooting** - Infrastructure vs application layer problem-solving
11. **Public IP Reliability** - Instance lifecycle behavior and Elastic IP management
12. **"My IP" Reliability** - Dynamic IP management and AWS behavior patterns
13. **Identity Center User Specification** - Federated identity and centralized access management
14. **Multiple Public IPs Architecture** - Advanced network interface implementation

**Topic 1: CIDR IP Range Understanding**

**Investigation Process**

Focused on understanding the mathematical foundation first. I learned that CIDR notation represents how many bits are "fixed" for the network portion versus "variable" for host addresses. Practiced calculating IP ranges using the formula: /16 means 16 bits for network, 16 bits for hosts = 2^16 = 65,536 total addresses.

**My Learning**

**The Mathematical Foundation:** When understood that /24 means the first 24 bits are fixed (network) and last 8 bits are variable (hosts). This means 2^8 = 256 total addresses minus 5 AWS reserved = 251 usable IPs. This mathematical approach made CIDR logical.

**Implementation**

Designed a practice VPC architecture for a scenario requiring web servers (100 IPs), app servers (50 IPs), and database servers (20 IPs). I allocated /24 subnets systematically:

* VPC: 10.0.0.0/16 (65,534 usable IPs total)
* Web Tier: 10.0.0.0/24 (254 IPs) - sufficient for 100 servers plus growth
* App Tier: 10.0.1.0/24 (254 IPs) - sufficient for 50 servers plus scaling
* DB Tier: 10.0.2.0/24 (254 IPs) - sufficient for 20 databases plus redundancy

| **CIDR** | **My Understanding** | **Usable IPs** | **Use Case** |
| --- | --- | --- | --- |
| /28 | 28 bits network, 4 bits hosts | 14 | Small database cluster |
| /24 | 24 bits network, 8 bits hosts | 254 | Standard application tier |
| /16 | 16 bits network, 16 bits hosts | 65,534 | Entire company VPC |

**Topic 2: Public vs Private Subnets**

**Subnet Types**

**Public Subnet:** Has a route to Internet Gateway (0.0.0.0/0 → IGW), which means resources can be directly accessed from the internet

**Private Subnet:** Only has routes to NAT Gateway for outbound access, meaning no direct inbound internet connectivity

**Security Principle:** This creates defense in depth through network layer isolation

**Example**

Designed a three-tier architecture where:

* **Public Subnets:** Hold load balancers and web servers that need internet access
* **Private App Subnets:** Hold application servers that only need outbound internet for updates
* **Private DB Subnets:** Hold databases with no internet access at all

**Topic 3: MFA + CLI Security - How I Understood Enterprise Patterns**

**Understanding**

Two MFA codes are required when adding an MFA device. This is related to TOTP (Time-based One-Time Password) synchronization:

* **First Code:** Establishes device timing baseline with AWS servers.
* **Second Code:** Validates time synchronization accuracy to prevent clock drift.
* **Security Purpose:** Prevents replay attacks and ensures reliable MFA operation

**Why AdminRole-MFA Approach is Secure**

Learnt that this implementation represents enterprise-grade security because:

* Base user has limited permissions (principle of least privilege)
* Role requires MFA device (something you have authentication factor)
* Temporary credentials with limited session duration (1-12 hours)
* Even if base credentials are compromised, no admin access without MFA device

# The secure pattern mastered:

aws sts assume-role \ --role-arn "arn:aws:iam::733366527973:role/AdminRole-MFA" \ --role-session-name "secure-session" \ --serial-number "arn:aws:iam::733366527973:mfa/device-1" \ --token-code [6-DIGIT-MFA-CODE] \ --profile admin-base

# This gives me temporary credentials that expire automatically

**Topic 4: Breakglass Users**

**What I Researched**

Investigated the concept of "breakglass" users and discovered the etymology comes from emergency "break glass" fire alarm boxes. These are special administrative accounts designed for emergency access when normal authentication systems fail.

**Emergency Scenarios I Identified**

Learnet breakglass accounts are needed when:

* **MFA Device Lost/Broken:** Primary admin can't access their MFA device
* **Identity Provider Outage:** SAML/SSO systems are unavailable
* **Mass Account Lockout:** Security incident locks all regular users
* **Automation Failure:** Infrastructure as Code systems break

**How To Implement This**

A proper breakglass account should have:

* **Separate email address** (not tied to corporate email system)
* **Physical MFA token** (not phone app that could be lost)
* **Full administrative permissions** but minimal usage pattern
* **Comprehensive monitoring** with immediate alerting on any usage
* **Regular testing** (monthly verification that it works)

**Business Continuity Value**

Understood that breakglass accounts support compliance frameworks like SOX by ensuring business continuity during emergencies while maintaining audit trails.

**Topic 5: PEM File Security – Restricting user permissions.**

**Security Strategy**

Discovered an approach where the compromised user can still login with the same credentials but loses administrative powers without knowing it:

# Silent privilege removal - no alerts to attacker

sudo deluser ec2-user sudo

# Remove admin privileges silently

sudo deluser ec2-user wheel

# Remove from admin groups

sudo deluser ec2-user admin

# Create secret admin backdoor for legitimate access

sudo useradd -m admin-backup

sudo usermod -aG sudo admin-backup

# Set up new SSH key for secure admin access

**What This Achieves**

**Compromised User Experience:**  
• Same SSH login still works  
• Gets "sudo: permission denied" errors  
• Likely thinks: "System issue, sudo broken"

**My Administrative Control:**  
• Full admin access via secret account  
• Complete monitoring of their activity  
• Time to plan proper response

**Topic 6: Security Groups vs NACLs role in Port Security.**

**Learning**

Think of Security Groups as doors with locks: each port is a different type of lock, and rules determine who gets keys to which locks. This made the security implications clear - opening port 22 to 0.0.0.0/0 means anyone on the internet can try to SSH into my server.

**Dangerous Configurations**

Learned to identify critical security risks:

* **Port 22 (SSH) open to 0.0.0.0/0:** Vulnerable to brute force attacks from anywhere
* **Port 3306 (MySQL) open to public:** Database directly accessible from internet
* **Port 3389 (RDP) open to world:** Windows servers exposed to remote attacks

**Security Audit Commands I Mastered**

# Find SSH open to everyone (CRITICAL DANGER!)

aws ec2 describe-security-groups --query 'SecurityGroups[?IpPermissions[?FromPort==`22` && IpRanges[?CidrIp==`0.0.0.0/0`]]].[GroupId,GroupName]'

# Find databases open to internet (IMMEDIATE FIX NEEDED!)

aws ec2 describe-security-groups --query 'SecurityGroups[?IpPermissions[?FromPort==`3306` && IpRanges[?CidrIp==`0.0.0.0/0`]]].[GroupId,GroupName]'

# Fix dangerous SSH rule

aws ec2 revoke-security-group-ingress --group-id sg-xxxxx --protocol tcp --port 22 --cidr 0.0.0.0/0

aws ec2 authorize-security-group-ingress --group-id sg-xxxxx --protocol tcp --port 22 --cidr MY\_OFFICE\_IP/32

**Security Rule**

Developed a simple principle: "If it's not a website (port 80/443), it should NOT be open to 0.0.0.0/0 (everyone)!" This helps to quickly identify dangerous configurations and security risks to others.

**Topic 7: Network Interfaces**

**Learning**

Different network interfaces like physical network cards in a computer:

* laptop has WiFi card = Network Interface #1 (connects to wireless networks)
* laptop has Ethernet port = Network Interface #2 (connects to wired networks)
* Each can connect to different networks with different IP addresses
* EC2 instance = Virtual computer with virtual network cards (ENIs)

**Critical Distinction**

Discovered that subnet ≠ network interface:

* **Subnet:** The "neighborhood" or network area
* **Network Interface:** The "network card" that connects TO the neighborhood
* **Key Point:** Interface goes INTO a subnet, not the other way around

**Access Misconception**

Learned that creating a network interface in the same subnet as an EC2 instance does NOT automatically provide access to that instance. The interface must be ATTACHED to the instance first, otherwise it's just "floating" with an IP address but connecting to nothing.

**Subnet Capacity Planning**

For calculating subnet capacity: /24 subnet = 256 total IPs - 5 AWS reserved = 251 available for network interfaces. Understood that web servers need larger subnets (/22) while databases can use smaller subnets (/28) for controlled capacity.

**Topic 8: Multiple Public IPs for an EC2 instance.**

**What I Discovered**

I learned that multiple public IPs for a single EC2 instance are possible through multiple network interfaces (ENIs). Each interface can have its own Elastic IP, allowing one instance to be accessible via multiple public IP addresses.

**Implementation Process**

1. Create additional network interface in target subnet
2. Attach interface to EC2 instance (device-index 1, 2, etc.)
3. Allocate Elastic IP from AWS
4. Associate Elastic IP with new network interface
5. Result: Same instance accessible via multiple public IPs

**Topic 9: Route Table Architecture - Learned Network Design Principles**

**What was Concluded**

**AWS Technical Capability:** ✅ Technically Valid - No AWS limits prevent 50 subnets sharing 1 route table

**Security Assessment:** ❌ Poor Practice - Eliminates network segmentation benefits

**Enterprise Readiness:** ❌ Not Recommended - Fails network architecture best practices

**Problems Identified**

Learned that using a single route table for 50 or more subnets creates several issues:

* **No Network Segmentation:** Database subnets get same routes as web subnets
* **Security Risk:** No network-level isolation between tiers
* **Management Complexity:** Single change affects all 50 subnets
* **Compliance Issues:** Violates network isolation requirements

**Topic 10: HTTP Troubleshooting - My Problem-Solving Success**

**Problem Encountered**

Had an HTTP inbound rule configured in the security group allowing access from all IPs, but still couldn't access the EC2 instance via browser using the public IP.

**My Diagnostic Process**

Systematically checked the infrastructure components:

* Security group rules - ✅ Correct
* Public IP assignment - ✅ Assigned
* Subnet routing - ✅ Public subnet
* Instance status - ✅ Running

**Critical Realization**

Discovered that the problem wasn't in the infrastructure layer but in the application layer. Realized that no web server was installed/running on the EC2 instance.

**How I Solved It**

Installed and configured a web server:

# Web server installation and setup

sudo yum update -y

sudo yum install -y httpd

sudo systemctl start httpd

sudo systemctl enable httpd

# Verification I performed curl localhost

curl localhost:80

# Browser test: http://my-public-ip - SUCCESS!

**Topic 11: Public IP Reliability - Learned Instance Lifecycle Behavior**

**Behavior Patterns Discovered**

| **Operation Tested** | **Public IP Behaviour** | **Observed** |
| --- | --- | --- |
| Reboot | Stays the same | ✅ No change in IP |
| Stop/Start | Changes to new random IP | ❌ New IP assigned |
| With Elastic IP | Always stays the same | ✅ Persistent through all operations |

**Solution I Implemented**

For applications requiring consistent IP addresses, I learned to use Elastic IPs:

# Allocate static IP

aws ec2 allocate-address --domain vpc

# Associate with instance

aws ec2 associate-address --instance-id i-xxxxx --allocation-id eipalloc-xxxxx

# Now IP persists through stop/start operations

**Commands Used**

# Network Security Analysis

aws ec2 describe-security-groups --query 'SecurityGroups[?IpPermissions[?IpRanges[?CidrIp==`0.0.0.0/0`]]].[GroupId,GroupName]'

aws ec2 describe-subnets --query 'Subnets[\*].{SubnetId:SubnetId,CIDR:CidrBlock,AvailableIPs:AvailableIpAddressCount}'

# MFA and Secure Access

aws sts assume-role --role-arn arn:aws:iam::ACCOUNT:role/AdminRole-MFA --role-session-name session --serial-number arn:aws:iam::ACCOUNT:mfa/device --token-code 123456

# Network Interface Management

aws ec2 create-network-interface --subnet-id subnet-xxxxx

aws ec2 attach-network-interface --network-interface-id eni-xxxxx --instance-id i-xxxxx --device-index 1

# Security Remediation

aws ec2 revoke-security-group-ingress --group-id sg-xxxxx --protocol tcp --port 22 --cidr 0.0.0.0/0

aws ec2 authorize-security-group-ingress --group-id sg-xxxxx --protocol tcp --port 22 --cidr MY\_OFFICE\_IP/32

**Topic 12: "My IP" Reliability in Security Groups - Learned Dynamic IP Management**

**What is Researched and Discovered**

Investigated the AWS documentation and discovered that when we select "My IP" in the AWS Console, AWS automatically detects the current public IP address and creates a rule like:

Source: 203.0.113.45/32 (my current IP) Port: 22 (SSH) Protocol: TCP

**Critical Finding:** AWS DOES automatically insert current public IPv4 address when we select "My IP", but it does NOT automatically update the rule if the IP changes later.

**The Real Problem Identified**

If my IP changes, I lose access completely! AWS doesn't automatically update the existing rule. Here's what happens:

* **Day 1:** Select "My IP" → AWS detects 203.0.113.45 → Rule created: 203.0.113.45/32
* **Day 2:** ISP assigns new IP: 203.0.113.67 → Security Group still shows: 203.0.113.45/32
* **Result:** SSH access fails because my new IP isn't in the security group

**Solutions**

##### Method 1: Update via AWS Console

1. Login to AWS Console (if have access from new IP)
2. Go to EC2 → Security Groups
3. Edit inbound rules
4. Delete old IP rule, add new "My IP" rule

##### Method 2: Update via CLI (if available)

# Remove old IP rule

aws ec2 revoke-security-group-ingress --group-id sg-xxxxx --protocol tcp --port 22 --cidr 203.0.113.45/32

# Get my new IP and add rule

$newIP = (curl ifconfig.me).trim()

aws ec2 authorize-security-group-ingress --group-id sg-xxxxx --protocol tcp --port 22 --cidr "$newIP/32"

**Prevention Strategies**

* **Broader IP Range:** Use my ISP's subnet range instead of single IP (/24 instead of /32)
* **Multiple IPs:** Add home, office, and mobile hotspot IPs to security group
* **VPN with Static IP:** Use VPN service with dedicated IP address
* **Alternative Access Methods:** Always have SSM Session Manager as backup
* **Automation Scripts:** Create scripts to detect and update IP changes

***Key Insight:*** *The "My IP" feature is a one-time detection, not continuous monitoring. We need to plan for IP changes and have backup access methods ready.*

**Topic 13: Identity Center User Specification**

**Discovered About Identity Center**

**AWS Identity Center (formerly SSO):** Centralized identity management service that provides single sign-on access to AWS accounts and business applications without creating individual IAM users in each account.

**Business Problem It Solves:** Managing user access across multiple AWS accounts without the complexity of separate IAM users everywhere.

**What "Specify a User" Actually Means**

1. **Creating user identities** in a central identity store
2. **Assigning permission sets** (like role templates) to users
3. **Granting access** to specific AWS accounts
4. **Managing group memberships** for easier administration

**Two Implementation Approaches**

##### Method 1: Create Users Directly in Identity Center

# Enable Identity Center (one-time setup)

aws sso-admin create-instance --name "MyOrganization-SSO"

# Create a user directly

aws identitystore create-user --identity-store-id d-xxxxxxxxxx --user-name "john.doe" --display-name "John Doe" --emails '[{"value":"john.doe@company.com","type":"work","primary":true}]'

# Create permission set (like an IAM role template)

aws sso-admin create-permission-set --instance-arn arn:aws:sso:::instance/ssoins-xxxxxxxxxx --name "SecurityAnalyst"

# Assign permission set to user for specific account

aws sso-admin create-account-assignment --instance-arn arn:aws:sso:::instance/ssoins-xxxxxxxxxx --target-id 123456789012 --target-type AWS\_ACCOUNT --principal-type USER --principal-id xxxxxxxxxx

##### Method 2: Connect External Identity Provider

Connect to existing corporate directories like:

* Microsoft Active Directory
* Okta
* Azure AD
* Google Workspace

**Why This is Better Than Traditional IAM**

**Traditional IAM Problems:**  
• Separate IAM user per AWS account  
• Multiple passwords/keys to manage  
• Complex cross-account access  
• No centralized user management

**Identity Center Benefits:**  
• Single user identity across all accounts  
• SSO access to console and CLI  
• Centralized permission management  
• Corporate directory integration

**Real-World Scenario**

**Challenge:** Give security analyst access to 3 AWS accounts (Dev, Staging, Prod) with different permissions

**My Solution with Identity Center:**

1. Create one user: security.analyst@company.com
2. Create permission sets: SecurityAnalyst-ReadOnly, SecurityAnalyst-Limited, SecurityAnalyst-Full
3. Assign user to each account with appropriate permission set
4. User gets single sign-on portal with all accounts visible

**Result:** One identity, three different access levels, single sign-on experience

**Topic 14: Multiple Public IPs for EC2 - Complete Architecture Understanding**

**Comprehensive Understanding I Achieved**

**YES, EC2 instances can have multiple public IPs** through multiple network interfaces (ENIs). Each interface can have its own Elastic IP, allowing one instance to be accessible via multiple public IP addresses.

**Complete Implementation Process I Mastered**

# Step 1: Create additional network interfaces

aws ec2 create-network-interface --subnet-id subnet-12345678 --description "Interface for SSL cert 2"

aws ec2 create-network-interface --subnet-id subnet-12345678 --description "Interface for SSL cert 3"

# Step 2: Attach interfaces to instance

aws ec2 attach-network-interface --network-interface-id eni-87654321 --instance-id i-1234567890abcdef0 --device-index 1

aws ec2 attach-network-interface --network-interface-id eni-11223344 --instance-id i-1234567890abcdef0 --device-index 2

# Step 3: Allocate Elastic IPs

aws ec2 allocate-address --domain vpc

aws ec2 allocate-address --domain vpc

# Step 4: Associate Elastic IPs with interfaces

aws ec2 associate-address --network-interface-id eni-87654321 --allocation-id eipalloc-87654321

aws ec2 associate-address --network-interface-id eni-11223344 --allocation-id eipalloc-11223344

**Limitations and Costs**

| **Instance Type** | **Max Network Interfaces** | **Max Public IPs** | **Monthly Cost (Additional IPs)** |
| --- | --- | --- | --- |
| t2.micro | 2 | 2 | $3.60 (1 additional IP) |
| t3.small | 3 | 3 | $7.20 (2 additional IPs) |
| m5.large | 3 | 3 | $7.20 (2 additional IPs) |
| m5.xlarge | 4 | 4 | $10.80 (3 additional IPs) |

**AWS Backlog Clarifications - Complete Personal Learning Journey Documentation**

Week 2 Comprehensive Security Training | My Learning Process