Identity and Access Management (IAM) - Complete Implementation Guide

1. Core IAM Architecture

1.1 Zero-Trust Identity Framework

```python
```

```python
# iam_core.py - Zero-Trust IAM System
import hashlib
import secrets
import jwt
import pyotp
import bcrypt
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass
from enum import Enum
import asyncio
import aioredis
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2
from cryptography.hazmat.backends import default_backend

class AuthenticationLevel(Enum):
    NONE = 0
    BASIC = 1
    MFA = 2
    BIOMETRIC = 3
    HARDWARE_KEY = 4
    QUANTUM_SAFE = 5

@dataclass
class IdentityContext:
    user_id: str
    session_id: str
    ip_address: str
    device_fingerprint: str
    location: Dict
    auth_level: AuthenticationLevel
    risk_score: float
    permissions: List[str]
    last_activity: datetime
    session_expiry: datetime

class ZeroTrustIAM:
    def __init__(self):
        self.sessions = {}
        self.user_store = {}
        self.permission_matrix = {}
        self.risk_engine = RiskAssessmentEngine()
        self.audit_logger = AuditLogger()

    async def authenticate(self, credentials: Dict) -> Optional[IdentityContext]:
        """Multi-factor authentication with adaptive security"""

        # Step 1: Basic credential verification
        user = await self.verify_credentials(credentials)
        if not user:
            await self.audit_logger.log_failed_auth(credentials)
            return None

        # Step 2: Risk assessment
        risk_score = await self.risk_engine.assess_risk({
            'user': user,
            'ip': credentials.get('ip_address'),
            'device': credentials.get('device_fingerprint'),
            'location': credentials.get('location'),
            'time': datetime.utcnow()
        })

        # Step 3: Determine required authentication level
        required_auth_level = self.determine_auth_level(risk_score)

        # Step 4: Perform additional authentication if needed
        if required_auth_level >= AuthenticationLevel.MFA:
            if not await self.verify_mfa(user, credentials.get('mfa_token')):
                return None

        if required_auth_level >= AuthenticationLevel.BIOMETRIC:
            if not await self.verify_biometric(user, credentials.get('biometric_data')):
```

```
            return None

        # Step 5: Create identity context
        context = IdentityContext(
            user_id=user['id'],
            session_id=self.generate_session_id(),
            ip_address=credentials.get('ip_address'),
            device_fingerprint=credentials.get('device_fingerprint'),
            location=credentials.get('location', {}),
            auth_level=required_auth_level,
            risk_score=risk_score,
            permissions=await self.get_user_permissions(user['id']),
            last_activity=datetime.utcnow(),
            session_expiry=datetime.utcnow() + timedelta(hours=1)
        )

        # Step 6: Store session
        self.sessions[context.session_id] = context

        # Step 7: Log successful authentication
        await self.audit_logger.log_successful_auth(context)

        return context
```

1.2 Advanced Permission Management

python

```python
class PermissionManager:
    """Attribute-Based Access Control (ABAC) with dynamic permissions"""

    def __init__(self):
        self.policies = {}
        self.resource_registry = {}
        self.role_hierarchy = {}

    async def check_permission(
        self,
        context: IdentityContext,
        resource: str,
        action: str,
        environment: Dict = None
    ) -> Tuple[bool, str]:
        """
        Check if user has permission for action on resource
        Returns: (allowed: bool, reason: str)
        """

        # Build evaluation context
        eval_context = {
            'subject': {
                'id': context.user_id,
                'auth_level': context.auth_level.value,
                'risk_score': context.risk_score,
                'permissions': context.permissions,
                'ip': context.ip_address,
                'location': context.location
            },
            'resource': {
                'type': resource.split(':')[0],
                'id': resource,
                'owner': self.get_resource_owner(resource),
                'classification': self.get_resource_classification(resource)
            },
            'action': action,
            'environment': environment or {
                'time': datetime.utcnow(),
                'day_of_week': datetime.utcnow().weekday(),
                'network': self.detect_network_type(context.ip_address)
            }
        }

        # Evaluate all applicable policies
        applicable_policies = self.find_applicable_policies(eval_context)

        for policy in applicable_policies:
            decision = await self.evaluate_policy(policy, eval_context)

            if decision == 'DENY':
                return False, f"Denied by policy: {policy['name']}"
            elif decision == 'ALLOW':
                # Check additional constraints
                if await self.check_constraints(policy, eval_context):
                    return True, f"Allowed by policy: {policy['name']}"

        return False, "No applicable policy found"

    async def evaluate_policy(self, policy: Dict, context: Dict) -> str:
        """Evaluate ABAC policy with complex conditions"""

        # Check subject conditions
        if not self.match_conditions(policy['subject'], context['subject']):
            return 'NOT_APPLICABLE'

        # Check resource conditions
        if not self.match_conditions(policy['resource'], context['resource']):
            return 'NOT_APPLICABLE'

        # Check action conditions
        if policy['action'] != '*' and policy['action'] != context['action']:
            return 'NOT_APPLICABLE'
```

```python
    # Check environment conditions
    if 'environment' in policy:
        if not self.match_conditions(policy['environment'], context['environment']):
            return 'NOT_APPLICABLE'

    # Check obligations (additional requirements)
    if 'obligations' in policy:
        for obligation in policy['obligations']:
            if not await self.fulfill_obligation(obligation, context):
                return 'DENY'

    return policy['effect']  # ALLOW or DENY
```

1.3 Session Management & Token Security

```python
class SecureSessionManager:
    """Quantum-resistant session management"""

    def __init__(self):
        self.redis_client = None
        self.token_secret = secrets.token_bytes(64)
        self.refresh_secret = secrets.token_bytes(64)

    async def create_session(self, identity: IdentityContext) -> Dict:
        """Create secure session with JWT tokens"""

        # Generate access token
        access_token = jwt.encode({
            'user_id': identity.user_id,
            'session_id': identity.session_id,
            'auth_level': identity.auth_level.value,
            'permissions': identity.permissions,
            'exp': datetime.utcnow() + timedelta(minutes=15),
            'iat': datetime.utcnow(),
            'jti': secrets.token_urlsafe(32)  # JWT ID for revocation
        }, self.token_secret, algorithm='HS512')

        # Generate refresh token
        refresh_token = jwt.encode({
            'user_id': identity.user_id,
            'session_id': identity.session_id,
            'exp': datetime.utcnow() + timedelta(days=7),
            'iat': datetime.utcnow(),
            'jti': secrets.token_urlsafe(32)
        }, self.refresh_secret, algorithm='HS512')

        # Store session in Redis with TTL
        await self.redis_client.setex(
            f"session:{identity.session_id}",
            3600,  # 1 hour TTL
            json.dumps({
                'identity': identity.__dict__,
                'access_jti': access_token['jti'],
                'refresh_jti': refresh_token['jti'],
                'created_at': datetime.utcnow().isoformat()
            })
        )

        return {
            'access_token': access_token,
            'refresh_token': refresh_token,
            'expires_in': 900,  # 15 minutes
            'token_type': 'Bearer'
        }
```

2. Multi-Factor Authentication (MFA)

2.1 TOTP/HOTP Implementation

```python
```

```python
class MFAManager:
    """Time-based and HMAC-based One-Time Password"""

    def __init__(self):
        self.backup_codes = {}

    def generate_secret(self, user_id: str) -> str:
        """Generate user-specific TOTP secret"""
        secret = pyotp.random_base32()
        # Store encrypted secret
        self.store_encrypted_secret(user_id, secret)
        return secret

    def generate_qr_code(self, user_id: str, secret: str) -> str:
        """Generate QR code for authenticator apps"""
        totp_uri = pyotp.totp.TOTP(secret).provisioning_uri(
            name=user_id,
            issuer_name='CyberFortress'
        )
        # Generate QR code
        import qrcode
        qr = qrcode.QRCode(version=1, box_size=10, border=5)
        qr.add_data(totp_uri)
        qr.make(fit=True)

        # Return base64 encoded image
        from io import BytesIO
        import base64
        img = qr.make_image(fill_color="black", back_color="white")
        buf = BytesIO()
        img.save(buf, format='PNG')
        return base64.b64encode(buf.getvalue()).decode()

    def verify_totp(self, user_id: str, token: str) -> bool:
        """Verify TOTP token with time window"""
        secret = self.get_encrypted_secret(user_id)
        totp = pyotp.TOTP(secret)

        # Allow 1 time step before/after for clock skew
        return totp.verify(token, valid_window=1)

    def generate_backup_codes(self, user_id: str, count: int = 10) -> List[str]:
        """Generate one-time backup codes"""
        codes = []
        for _ in range(count):
            code = ''.join(secrets.choice('0123456789') for _ in range(8))
            # Format as XXXX-XXXX
            formatted = f"{code[:4]}-{code[4:]}"
            codes.append(formatted)

        # Store hashed backup codes
        self.backup_codes[user_id] = [
            bcrypt.hashpw(code.encode(), bcrypt.gensalt())
            for code in codes
        ]

        return codes
```

2.2 Biometric Authentication

```python
```

```python
class BiometricAuthenticator:
    """Advanced biometric authentication system"""

    def __init__(self):
        self.biometric_store = {}
        self.anti_spoofing = AntiSpoofingEngine()

    async def enroll_biometric(
        self,
        user_id: str,
        biometric_type: str,
        biometric_data: bytes
    ) -> bool:
        """Enroll user biometric data"""

        # Verify liveness/anti-spoofing
        if not await self.anti_spoofing.verify_liveness(biometric_data):
            raise SecurityError("Liveness check failed")

        # Extract biometric template
        template = await self.extract_template(biometric_type, biometric_data)

        # Encrypt and store template
        encrypted_template = await self.encrypt_biometric(template)

        self.biometric_store[user_id] = {
            'type': biometric_type,
            'template': encrypted_template,
            'enrolled_at': datetime.utcnow(),
            'quality_score': await self.assess_quality(template)
        }

        return True

    async def verify_biometric(
        self,
        user_id: str,
        biometric_data: bytes
    ) -> Tuple[bool, float]:
        """Verify biometric authentication"""

        if user_id not in self.biometric_store:
            return False, 0.0

        stored = self.biometric_store[user_id]

        # Extract template from provided data
        template = await self.extract_template(stored['type'], biometric_data)

        # Decrypt stored template
        stored_template = await self.decrypt_biometric(stored['template'])

        # Compare templates
        match_score = await self.compare_templates(template, stored_template)

        # Dynamic threshold based on security level
        threshold = self.get_dynamic_threshold(user_id)

        return match_score >= threshold, match_score
```

3. Privileged Access Management (PAM)

3.1 Just-In-Time Access

```python
```

```python
class PrivilegedAccessManager:
    """JIT access with automatic de-provisioning"""

    def __init__(self):
        self.elevated_sessions = {}
        self.approval_workflow = ApprovalWorkflow()

    async def request_privileged_access(
        self,
        requester: IdentityContext,
        resource: str,
        duration: int,
        justification: str
    ) -> str:
        """Request temporary privileged access"""

        request = {
            'id': secrets.token_urlsafe(16),
            'requester': requester.user_id,
            'resource': resource,
            'duration': duration,
            'justification': justification,
            'requested_at': datetime.utcnow(),
            'status': 'PENDING'
        }

        # Check if auto-approval is allowed
        if await self.can_auto_approve(requester, resource):
            return await self.grant_access(request)

        # Otherwise, start approval workflow
        await self.approval_workflow.submit(request)
        return request['id']

    async def grant_access(self, request: Dict) -> str:
        """Grant temporary elevated access"""

        # Create time-limited token
        token = jwt.encode({
            'request_id': request['id'],
            'user_id': request['requester'],
            'resource': request['resource'],
            'granted_at': datetime.utcnow().isoformat(),
            'expires_at': (datetime.utcnow() + timedelta(minutes=request['duration'])).isoformat(),
            'type': 'PRIVILEGED_ACCESS'
        }, self.token_secret, algorithm='HS512')

        # Schedule automatic revocation
        asyncio.create_task(
            self.schedule_revocation(request['id'], request['duration'])
        )

        # Log privileged access grant
        await self.audit_logger.log_privileged_grant(request)

        return token
```

3.2 Credential Vault

```python
```

```python
class SecureCredentialVault:
    """Hardware security module integrated credential vault"""

    def __init__(self):
        self.hsm_client = HSMClient()
        self.credentials = {}

    async def store_credential(
        self,
        credential_id: str,
        credential_data: Dict,
        owner: str,
        access_policy: Dict
    ) -> bool:
        """Store credential with encryption"""

        # Generate unique encryption key in HSM
        key_id = await self.hsm_client.generate_key(
            algorithm='AES256-GCM',
            extractable=False
        )

        # Encrypt credential data
        encrypted_data = await self.hsm_client.encrypt(
            key_id,
            json.dumps(credential_data).encode()
        )

        # Store encrypted credential
        self.credentials[credential_id] = {
            'encrypted_data': encrypted_data,
            'key_id': key_id,
            'owner': owner,
            'access_policy': access_policy,
            'created_at': datetime.utcnow(),
            'last_accessed': None,
            'access_count': 0
        }

        return True

    async def retrieve_credential(
        self,
        credential_id: str,
        requester: IdentityContext
    ) -> Optional[Dict]:
        """Retrieve credential with policy enforcement"""

        if credential_id not in self.credentials:
            return None

        cred = self.credentials[credential_id]

        # Check access policy
        if not await self.check_access_policy(cred['access_policy'], requester):
            await self.audit_logger.log_unauthorized_access(credential_id, requester)
            return None

        # Decrypt credential
        decrypted_data = await self.hsm_client.decrypt(
            cred['key_id'],
            cred['encrypted_data']
        )

        # Update access metadata
        cred['last_accessed'] = datetime.utcnow()
        cred['access_count'] += 1

        # Log access
        await self.audit_logger.log_credential_access(credential_id, requester)

        return json.loads(decrypted_data)
```

4. Identity Federation & SSO

4.1 SAML 2.0 Implementation

```python
class SAMLIdentityProvider:
    """SAML 2.0 Identity Provider"""

    def __init__(self):
        self.metadata = self.generate_idp_metadata()
        self.service_providers = {}

    def generate_saml_response(
        self,
        user: IdentityContext,
        sp_entity_id: str,
        acs_url: str
    ) -> str:
        """Generate signed SAML response"""

        from lxml import etree
        from signxml import XMLSigner

        # Create SAML response
        response = etree.Element(
            '{urn:oasis:names:tc:SAML:2.0:protocol}Response',
            ID=f"_{secrets.token_hex(16)}",
            Version="2.0",
            IssueInstant=datetime.utcnow().isoformat(),
            Destination=acs_url
        )

        # Add Issuer
        issuer = etree.SubElement(
            response,
            '{urn:oasis:names:tc:SAML:2.0:assertion}Issuer'
        )
        issuer.text = self.metadata['entity_id']

        # Add Status
        status = etree.SubElement(
            response,
            '{urn:oasis:names:tc:SAML:2.0:protocol}Status'
        )
        status_code = etree.SubElement(
            status,
            '{urn:oasis:names:tc:SAML:2.0:protocol}StatusCode',
            Value='urn:oasis:names:tc:SAML:2.0:status:Success'
        )

        # Create Assertion
        assertion = self.create_assertion(user, sp_entity_id)
        response.append(assertion)

        # Sign the response
        signer = XMLSigner(
            method=signxml.methods.enveloped,
            signature_algorithm='rsa-sha256'
        )
        signed_response = signer.sign(
            response,
            key=self.private_key,
            cert=self.certificate
        )

        # Base64 encode
        import base64
        return base64.b64encode(
            etree.tostring(signed_response)
        ).decode()
```

4.2 OAuth 2.0 / OpenID Connect

python

```python
class OIDCProvider:
    """OpenID Connect Provider with PKCE"""

    def __init__(self):
        self.clients = {}
        self.authorization_codes = {}
        self.access_tokens = {}

    async def authorize(
        self,
        client_id: str,
        redirect_uri: str,
        scope: str,
        state: str,
        code_challenge: str,
        code_challenge_method: str = 'S256'
    ) -> str:
        """OAuth 2.0 authorization endpoint with PKCE"""

        # Validate client
        if client_id not in self.clients:
            raise ValueError("Invalid client_id")

        client = self.clients[client_id]

        # Validate redirect URI
        if redirect_uri not in client['redirect_uris']:
            raise ValueError("Invalid redirect_uri")

        # Generate authorization code
        code = secrets.token_urlsafe(32)

        # Store code with PKCE challenge
        self.authorization_codes[code] = {
            'client_id': client_id,
            'redirect_uri': redirect_uri,
            'scope': scope,
            'state': state,
            'code_challenge': code_challenge,
            'code_challenge_method': code_challenge_method,
            'created_at': datetime.utcnow(),
            'expires_at': datetime.utcnow() + timedelta(minutes=10)
        }

        return code

    async def token(
        self,
        grant_type: str,
        code: str = None,
        refresh_token: str = None,
        code_verifier: str = None
    ) -> Dict:
        """Token endpoint with PKCE verification"""

        if grant_type == 'authorization_code':
            # Validate authorization code
            if code not in self.authorization_codes:
                raise ValueError("Invalid authorization code")

            code_data = self.authorization_codes[code]

            # Check expiration
            if datetime.utcnow() > code_data['expires_at']:
                raise ValueError("Authorization code expired")

            # Verify PKCE
            if code_data['code_challenge_method'] == 'S256':
                import hashlib
                challenge = base64.urlsafe_b64encode(
                    hashlib.sha256(code_verifier.encode()).digest()
                ).decode().rstrip('=')
```

```
            if challenge != code_data['code_challenge']:
                raise ValueError("Invalid code_verifier")

        # Generate tokens
        access_token = jwt.encode({
            'sub': user_id,
            'client_id': code_data['client_id'],
            'scope': code_data['scope'],
            'exp': datetime.utcnow() + timedelta(hours=1),
            'iat': datetime.utcnow(),
            'jti': secrets.token_urlsafe(16)
        }, self.token_secret, algorithm='RS256')

        # Generate ID token for OpenID Connect
        id_token = jwt.encode({
            'iss': self.metadata['issuer'],
            'sub': user_id,
            'aud': code_data['client_id'],
            'exp': datetime.utcnow() + timedelta(hours=1),
            'iat': datetime.utcnow(),
            'nonce': code_data.get('nonce')
        }, self.token_secret, algorithm='RS256')

        # Delete used authorization code
        del self.authorization_codes[code]

        return {
            'access_token': access_token,
            'token_type': 'Bearer',
            'expires_in': 3600,
            'id_token': id_token,
            'scope': code_data['scope']
        }
```

5. Continuous Authentication & Risk Assessment

5.1 Behavioral Analytics

```
python
```

```python
class BehavioralAnalyticsEngine:
    """User behavior analytics for continuous authentication"""

    def __init__(self):
        self.user_profiles = {}
        self.ml_model = self.load_ml_model()

    async def analyze_behavior(
        self,
        user_id: str,
        action: Dict
    ) -> float:
        """Analyze user behavior and return risk score"""

        # Get user's behavioral profile
        profile = self.user_profiles.get(user_id, self.create_default_profile())

        # Extract features
        features = self.extract_features(action, profile)

        # Calculate anomaly score using ML model
        anomaly_score = self.ml_model.predict_proba([features])[0][1]

        # Update behavioral profile
        await self.update_profile(user_id, action, anomaly_score)

        # Calculate risk score
        risk_score = self.calculate_risk_score(
            anomaly_score,
            action,
            profile
        )

        # Trigger re-authentication if needed
        if risk_score > 0.8:
            await self.trigger_step_up_auth(user_id, risk_score)

        return risk_score

    def extract_features(self, action: Dict, profile: Dict) -> List[float]:
        """Extract behavioral features for ML model"""

        features = []

        # Time-based features
        features.append(self.get_time_deviation(action['timestamp'], profile))
        features.append(self.get_day_of_week_score(action['timestamp'], profile))

        # Location features
        features.append(self.get_location_deviation(action['location'], profile))
        features.append(self.get_velocity_score(action['location'], profile))

        # Action pattern features
        features.append(self.get_action_frequency_score(action['type'], profile))
        features.append(self.get_action_sequence_score(action['type'], profile))

        # Device features
        features.append(self.get_device_trust_score(action['device'], profile))

        # Network features
        features.append(self.get_network_reputation(action['ip_address']))

        return features
```

6. Audit & Compliance

6.1 Comprehensive Audit Logging

python

```python
class AuditLogger:
    """Tamper-proof audit logging system"""

    def __init__(self):
        self.blockchain_logger = BlockchainAuditLogger()
        self.siem_integration = SIEMIntegration()

    async def log_event(
        self,
        event_type: str,
        actor: IdentityContext,
        resource: str,
        action: str,
        result: str,
        metadata: Dict = None
    ):
        """Log security event with blockchain anchoring"""

        event = {
            'id': secrets.token_urlsafe(16),
            'timestamp': datetime.utcnow().isoformat(),
            'event_type': event_type,
            'actor': {
                'user_id': actor.user_id,
                'session_id': actor.session_id,
                'ip_address': actor.ip_address,
                'auth_level': actor.auth_level.value
            },
            'resource': resource,
            'action': action,
            'result': result,
            'metadata': metadata or {},
            'integrity_hash': None
        }

        # Calculate integrity hash
        event['integrity_hash'] = self.calculate_integrity_hash(event)

        # Store in immutable log
        await self.store_immutable(event)

        # Anchor to blockchain for tamper-proof evidence
        await self.blockchain_logger.anchor(event['integrity_hash'])

        # Send to SIEM
        await self.siem_integration.send(event)

        # Check for compliance violations
        await self.check_compliance(event)
```

7. Implementation Checklist

Phase 1: Core IAM (Week 1-2)

- [ ] Implement zero-trust authentication framework
- [ ] Set up secure session management
- [ ] Deploy Redis for session storage
- [ ] Implement basic RBAC

Phase 2: MFA & Advanced Auth (Week 3-4)

- [ ] Implement TOTP/HOTP
- [ ] Add backup codes system
- [ ] Integrate biometric authentication
- [ ] Deploy risk assessment engine

Phase 3: PAM & Federation (Week 5-6)

- [ ] Implement JIT access
- [ ] Deploy credential vault
- [ ] Set up SAML 2.0
- [ ] Implement OAuth/OIDC

Phase 4: Analytics & Audit (Week 7-8)

- [ ] Deploy behavioral analytics
- [ ] Implement continuous authentication
- [ ] Set up audit logging
- [ ] Integrate with SIEM

8. Security Best Practices

1. **Password Policy**:
   - Minimum 14 characters
   - Require complexity
   - Check against breach databases
   - Enforce regular rotation for privileged accounts
2. **Session Security**:
   - Use secure, httpOnly, sameSite cookies
   - Implement CSRF protection
   - Rotate session IDs after login
   - Implement idle timeout
3. **Encryption**:
   - Use AES-256-GCM for data at rest
   - TLS 1.3 for data in transit
   - Quantum-resistant algorithms ready
   - Hardware security module for key storage
4. **Monitoring**:
   - Real-time alerting for suspicious activity
   - Failed login tracking
   - Privilege escalation monitoring
   - Geographic anomaly detection