Module 6: Security & Authentication

Master API keys, OAuth, and secure communication systems for MEV operations

Course Overview

Duration: 130 minutes
Level: Advanced
Prerequisites: Modules 1-5, understanding of cybersecurity and authentication systems
Certificate: Available upon completion

Learning Objectives

By the end of this module, you will be able to:
- Design comprehensive authentication and authorization systems for MEV APIs
- Implement secure API key management and rotation policies
- Build OAuth 2.0 and OpenID Connect integrations for user authentication
- Create secure communication channels with encryption and certificate management
- Implement multi-factor authentication and access control systems
- Design compliance frameworks and audit logging systems for MEV operations

---

Table of Contents

---

1. Security Architecture Design

Security Framework for MEV Systems

```yaml
# security/security-framework.yml
apiVersion: v1
kind: ConfigMap
metadata:
  name: security-config
  namespace: mev-system
data:
  security-policy.yaml: |
    # MEV System Security Policy
    version: "1.0"
    policies:
      authentication:
        - name: "Strong Password Policy"
          rules:
            - min_length: 12
              require_uppercase: true
              require_lowercase: true
              require_numbers: true
              require_symbols: true
              prevent_common_passwords: true
              password_history: 5
              max_age_days: 90

        - name: "Multi-Factor Authentication"
          requirements:
            - mfa_required: true
            - backup_codes: true
            - biometric_support: true
            - hardware_token_support: true

        - name: "Session Management"
          rules:
            - max_session_duration: 8h
            - idle_timeout: 30m
            - concurrent_sessions: 3
            - secure_cookies: true
            - httponly_cookies: true
            - samesite_strict: true

      authorization:
        - name: "Role-Based Access Control"
          roles:
```

```
        - name: "admin"
          permissions: ["*"]
          restrictions: ["mev-trading", "funds-management"]
        - name: "trader"
          permissions: ["mev-view", "mev-execute"]
          restrictions: ["funds-management"]
        - name: "viewer"
          permissions: ["mev-view"]
          restrictions: []
        - name: "service-account"
          permissions: ["mev-api"]
          restrictions: ["web-access"]

    - name: "API Rate Limiting"
      rules:
        - endpoint: "/api/v1/execute"
          limit: 10/minute
          burst: 5
        - endpoint: "/api/v1/opportunities"
          limit: 100/minute
          burst: 20
        - endpoint: "/api/v1/metrics"
          limit: 60/minute
          burst: 10

  encryption:
    - name: "Data Encryption at Rest"
      algorithms:
        - AES-256-GCM
        - ChaCha20-Poly1305
      key_rotation: 90d
    - name: "Data Encryption in Transit"
      protocols:
        - TLS_1.3
        - HTTPS_ONLY
      certificate_requirements:
        - min_key_size: 2048
        - signature_algorithm: SHA-256
        - key_usage: [digitalSignature, keyEncipherment]
        - san_dns: ["*.mev-system.com"]

  audit:
```

```
        - name: "Audit Logging"
          events:
            - authentication_attempts
            - authorization_failures
            - data_access
            - configuration_changes
            - system_events
          retention: 7y
          encryption: true
          integrity_protection: true
        - name: "Compliance Requirements"
          standards:
            - SOC_2_Type_II
            - ISO_27001
            - PCI_DSS
            - GDPR
            - CCPA
```

Security Architecture Diagram

```python
# security/architecture.py
from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Any, Union
from dataclasses import dataclass, field
from enum import Enum
import asyncio
import logging
from datetime import datetime, timedelta
import hashlib
import secrets
import json

class AuthMethod(Enum):
    API_KEY = "api_key"
    OAUTH2 = "oauth2"
    JWT = "jwt"
    MUTUAL_TLS = "mutual_tls"
    BASIC_AUTH = "basic_auth"

class Permission(Enum):
    MEV_VIEW = "mev:view"
    MEV_EXECUTE = "mev:execute"
    MEV_ADMIN = "mev:admin"
    FUNDS_MANAGE = "funds:manage"
    SYSTEM_CONFIG = "system:config"
    AUDIT_VIEW = "audit:view"

class RiskLevel(Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"

@dataclass
class User:
    """User account with security attributes"""
    user_id: str
    username: str
    email: str
    roles: List[str]
    permissions: List[Permission]
    mfa_enabled: bool = False
```

```
    last_login: Optional[datetime] = None
    failed_login_attempts: int = 0
    account_locked: bool = False
    created_at: datetime = field(default_factory=datetime.utcnow)
    updated_at: Optional[datetime] = None

    def has_permission(self, permission: Permission) -> bool:
        """Check if user has specific permission"""
        return permission in self.permissions

    def has_role(self, role: str) -> bool:
        """Check if user has specific role"""
        return role in self.roles

@dataclass
class APIKey:
    """API key with security metadata"""
    key_id: str
    user_id: str
    key_hash: str  # Hashed version for storage
    key_prefix: str  # First few characters for identification
    permissions: List[Permission]
    rate_limit: int
    expires_at: Optional[datetime]
    last_used: Optional[datetime]
    created_at: datetime = field(default_factory=datetime.utcnow)
    usage_count: int = 0

    @classmethod
    def generate(cls, user_id: str, permissions: List[Permission]) ->
'APIKey':
        """Generate new API key"""
        # Generate secure random key
        raw_key = secrets.token_urlsafe(32)
        key_hash = hashlib.sha256(raw_key.encode()).hexdigest()
        key_prefix = raw_key[:8]

        return cls(
            key_id=secrets.token_urlsafe(16),
            user_id=user_id,
            key_hash=key_hash,
            key_prefix=key_prefix,
```

```
            permissions=permissions,
            rate_limit=1000,
            expires_at=datetime.utcnow() + timedelta(days=90)
        )

    def verify_key(self, provided_key: str) -> bool:
        """Verify provided API key against stored hash"""
        if self.expires_at and datetime.utcnow() > self.expires_at:
            return False

        provided_hash =
hashlib.sha256(provided_key.encode()).hexdigest()
        return provided_hash == self.key_hash

@dataclass
class SecurityEvent:
    """Security event for audit logging"""
    event_id: str
    event_type: str
    user_id: Optional[str]
    source_ip: str
    user_agent: str
    severity: RiskLevel
    timestamp: datetime = field(default_factory=datetime.utcnow)
    details: Dict[str, Any] = field(default_factory=dict)
    resolved: bool = False

    def to_dict(self) -> Dict[str, Any]:
        return {
            'event_id': self.event_id,
            'event_type': self.event_type,
            'user_id': self.user_id,
            'source_ip': self.source_ip,
            'user_agent': self.user_agent,
            'severity': self.severity.value,
            'timestamp': self.timestamp.isoformat(),
            'details': self.details,
            'resolved': self.resolved
        }

class SecurityManager:
    """Central security manager for MEV system"""
```

```
    def __init__(self):
        self.logger = logging.getLogger(__name__)

        # User and API key storage (in production, use database)
        self.users: Dict[str, User] = {}
        self.api_keys: Dict[str, APIKey] = {}
        self.security_events: List[SecurityEvent] = []

        # Rate limiting
        self.rate_limiters: Dict[str, Dict] = {}

        # Security configuration
        self.config = self._load_security_config()

        # Initialize default users
        self._initialize_default_users()

    def _load_security_config(self) -> Dict[str, Any]:
        """Load security configuration"""
        return {
            'max_failed_login_attempts': 5,
            'lockout_duration_minutes': 30,
            'password_min_length': 12,
            'api_key_expiry_days': 90,
            'session_timeout_minutes': 480,
            'mfa_required_roles': ['admin', 'trader'],
            'rate_limit_default': 1000,
            'encryption_algorithm': 'AES-256-GCM',
            'audit_retention_days': 2555  # 7 years
        }

    def _initialize_default_users(self):
        """Initialize default system users"""

        # Admin user
        admin_user = User(
            user_id="admin-001",
            username="admin",
            email="admin@mev-system.com",
            roles=["admin"],
            permissions=list(Permission),
```

```
            mfa_enabled=True
        )
        self.users["admin-001"] = admin_user

        # Service account
        service_user = User(
            user_id="service-001",
            username="mev-service",
            email="service@mev-system.com",
            roles=["service-account"],
            permissions=[Permission.MEV_VIEW, Permission.MEV_EXECUTE],
            mfa_enabled=False
        )
        self.users["service-001"] = service_user

        self.logger.info("Initialized default users")

    async def authenticate_user(self,
                                username: str,
                                password: str,
                                source_ip: str,
                                user_agent: str) -> Optional[User]:
        """Authenticate user with username/password"""

        # Find user
        user = None
        for u in self.users.values():
            if u.username == username:
                user = u
                break

        if not user:
            await self._log_security_event(
                "AUTHENTICATION_FAILED", None, source_ip, user_agent,
                RiskLevel.MEDIUM, {"reason": "user_not_found",
"username": username}
            )
            return None

        # Check if account is locked
        if user.account_locked:
            await self._log_security_event(
```

```
                "AUTHENTICATION_BLOCKED", user.user_id, source_ip,
user_agent,
                RiskLevel.HIGH, {"reason": "account_locked"}
            )
            return None

        # Verify password (simplified - in production use proper
hashing)
        if not await self._verify_password(password, user):
            user.failed_login_attempts += 1

            # Check if should lock account
            if user.failed_login_attempts >=
self.config['max_failed_login_attempts']:
                user.account_locked = True
                await self._log_security_event(
                    "ACCOUNT_LOCKED", user.user_id, source_ip,
user_agent,
                    RiskLevel.HIGH, {"attempts":
user.failed_login_attempts}
                )
            else:
                await self._log_security_event(
                    "AUTHENTICATION_FAILED", user.user_id, source_ip,
user_agent,
                    RiskLevel.MEDIUM, {"attempts":
user.failed_login_attempts}
                )

            return None

        # Successful authentication
        user.last_login = datetime.utcnow()
        user.failed_login_attempts = 0

        await self._log_security_event(
            "AUTHENTICATION_SUCCESS", user.user_id, source_ip,
user_agent,
            RiskLevel.LOW, {}
        )

        return user
```

```
    async def _verify_password(self, password: str, user: User) ->
bool:
        """Verify password (simplified implementation)"""
        # In production, use proper password hashing like bcrypt or
argon2
        # This is a simplified check
        return password == "secure_password_123!"  # Placeholder

    async def generate_api_key(self,
                               user_id: str,
                               permissions: List[Permission],
                               rate_limit: Optional[int] = None) ->
Optional[str]:
        """Generate new API key for user"""

        if user_id not in self.users:
            return None

        # Create API key
        api_key = APIKey.generate(user_id, permissions)
        if rate_limit:
            api_key.rate_limit = rate_limit

        # Store API key
        self.api_keys[api_key.key_id] = api_key

        # Log security event
        await self._log_security_event(
            "API_KEY_CREATED", user_id, "system", "security-manager",
            RiskLevel.MEDIUM, {"key_id": api_key.key_id, "permissions":
[p.value for p in permissions]}
        )

        # Return the actual key (only shown once)
        return api_key.key_hash[:8] + api_key.key_id  # Simplified
return

    async def authenticate_api_key(self,
                                   provided_key: str,
                                   source_ip: str,
                                   user_agent: str) -> Optional[User]:
```

```
        """Authenticate API key"""

        # Find API key (simplified - in production use efficient
lookup)
        api_key = None
        for key in self.api_keys.values():
            if key.verify_key(provided_key):
                api_key = key
                break

        if not api_key:
            await self._log_security_event(
                "API_KEY_INVALID", None, source_ip, user_agent,
                RiskLevel.MEDIUM, {"provided_key": provided_key[:8] +
"..."}
            )
            return None

        # Check if key is expired
        if api_key.expires_at and datetime.utcnow() >
api_key.expires_at:
            await self._log_security_event(
                "API_KEY_EXPIRED", api_key.user_id, source_ip,
user_agent,
                RiskLevel.MEDIUM, {"key_id": api_key.key_id}
            )
            return None

        # Update usage
        api_key.usage_count += 1
        api_key.last_used = datetime.utcnow()

        # Get user
        user = self.users.get(api_key.user_id)

        await self._log_security_event(
            "API_KEY_AUTHENTICATED", api_key.user_id, source_ip,
user_agent,
            RiskLevel.LOW, {"key_id": api_key.key_id}
        )

        return user
```

```
    async def check_permission(self,
                               user: User,
                               permission: Permission,
                               context: Optional[Dict[str, Any]] = None)
-> bool:
        """Check if user has permission for action"""

        has_permission = user.has_permission(permission)

        # Log security event for permission checks
        if not has_permission:
            await self._log_security_event(
                "PERMISSION_DENIED", user.user_id, "system", "auth-
check",
                RiskLevel.MEDIUM, {
                    "permission": permission.value,
                    "user_roles": user.roles,
                    "context": context or {}
                }
            )

        return has_permission

    async def check_rate_limit(self,
                               user_id: str,
                               endpoint: str,
                               window_seconds: int = 60) -> bool:
        """Check if user is within rate limits"""

        key = f"{user_id}:{endpoint}"
        current_time = datetime.utcnow()

        if key not in self.rate_limiters:
            self.rate_limiters[key] = {"count": 0, "window_start":
current_time}

        limiter = self.rate_limiters[key]

        # Reset window if needed
        if (current_time - limiter["window_start"]).total_seconds() >
window_seconds:
```

```
            limiter["count"] = 0
            limiter["window_start"] = current_time

        # Get user's API key for rate limit
        user_key = None
        for key_obj in self.api_keys.values():
            if key_obj.user_id == user_id:
                user_key = key_obj
                break

        if user_key:
            limit = user_key.rate_limit
        else:
            limit = self.config['rate_limit_default']

        if limiter["count"] >= limit:
            await self._log_security_event(
                "RATE_LIMIT_EXCEEDED", user_id, "system", "rate-
limiter",
                RiskLevel.MEDIUM, {
                    "endpoint": endpoint,
                    "count": limiter["count"],
                    "limit": limit
                }
            )
            return False

        limiter["count"] += 1
        return True

    async def _log_security_event(self,
                                  event_type: str,
                                  user_id: Optional[str],
                                  source_ip: str,
                                  user_agent: str,
                                  severity: RiskLevel,
                                  details: Dict[str, Any]):
        """Log security event for audit"""

        event = SecurityEvent(
            event_id=secrets.token_urlsafe(16),
            event_type=event_type,
```

```
            user_id=user_id,
            source_ip=source_ip,
            user_agent=user_agent,
            severity=severity,
            details=details
        )

        self.security_events.append(event)

        # Log to application logger
        self.logger.info(f"Security event: {event_type} - User:
{user_id} - Severity: {severity.value}")

        # In production, also write to secure audit log
        await self._write_audit_log(event)

    async def _write_audit_log(self, event: SecurityEvent):
        """Write event to secure audit log (implementation depends on
system)"""
        # In production, write to encrypted, tamper-proof audit log
        pass

    async def get_security_events(self,
                                  user_id: Optional[str] = None,
                                  event_type: Optional[str] = None,
                                  limit: int = 100) ->
List[SecurityEvent]:
        """Get security events for audit"""

        filtered_events = self.security_events

        if user_id:
            filtered_events = [e for e in filtered_events if e.user_id
== user_id]

        if event_type:
            filtered_events = [e for e in filtered_events if
e.event_type == event_type]

        # Return most recent events
        return sorted(filtered_events, key=lambda x: x.timestamp,
reverse=True)[:limit]
```

```
    def get_user_statistics(self, user_id: str) -> Dict[str, Any]:
        """Get security statistics for user"""

        user_events = [e for e in self.security_events if e.user_id ==
user_id]

        return {
            "total_events": len(user_events),
            "failed_logins": len([e for e in user_events if
e.event_type == "AUTHENTICATION_FAILED"]),
            "successful_logins": len([e for e in user_events if
e.event_type == "AUTHENTICATION_SUCCESS"]),
            "permission_denials": len([e for e in user_events if
e.event_type == "PERMISSION_DENIED"]),
            "api_key_usage": len([e for e in user_events if
e.event_type == "API_KEY_AUTHENTICATED"]),
            "last_activity": max([e.timestamp for e in user_events],
default=None)
        }

# Security middleware for API endpoints
class SecurityMiddleware:
    """Security middleware for API request processing"""

    def __init__(self, security_manager: SecurityManager):
        self.security_manager = security_manager
        self.logger = logging.getLogger(__name__)

    async def authenticate_request(self,
                                   request_headers: Dict[str, str],
                                   source_ip: str) -> Optional[User]:
        """Authenticate incoming API request"""

        # Check for API key
        api_key = request_headers.get('X-API-Key')
        if api_key:
            return await self.security_manager.authenticate_api_key(
                api_key, source_ip, request_headers.get('User-Agent',
'')
            )
```

```
        # Check for Bearer token (JWT)
        auth_header = request_headers.get('Authorization')
        if auth_header and auth_header.startswith('Bearer '):
            # In production, validate JWT token
            pass

        # Check for basic auth
        basic_auth = request_headers.get('Authorization')
        if basic_auth and basic_auth.startswith('Basic '):
            # In production, decode and validate basic auth
            pass

        return None

    async def authorize_request(self,
                                user: User,
                                endpoint: str,
                                method: str) -> bool:
        """Authorize API request"""

        # Map endpoint to required permission
        required_permission =
self._map_endpoint_to_permission(endpoint, method)

        if required_permission:
            return await self.security_manager.check_permission(user,
required_permission)

        return True

    def _map_endpoint_to_permission(self, endpoint: str, method: str) -
> Optional[Permission]:
        """Map API endpoint to required permission"""

        # Simplified mapping
        if '/execute' in endpoint:
            return Permission.MEV_EXECUTE
        elif '/admin' in endpoint:
            return Permission.MEV_ADMIN
        elif '/funds' in endpoint:
            return Permission.FUNDS_MANAGE
        elif '/metrics' in endpoint or '/opportunities' in endpoint:
```

```
            return Permission.MEV_VIEW

        return None
```

---

2. API Key Management

Advanced API Key Management System

```python
# security/api_key_manager.py
import secrets
import hashlib
import hmac
from typing import Dict, List, Optional, Tuple
from datetime import datetime, timedelta
from dataclasses import dataclass, field
from enum import Enum
import json
import asyncio

class KeyStatus(Enum):
    ACTIVE = "active"
    EXPIRED = "expired"
    REVOKED = "revoked"
    SUSPENDED = "suspended"

class KeyUsage(Enum):
    READ_ONLY = "read_only"
    READ_WRITE = "read_write"
    EXECUTE_ONLY = "execute_only"
    FULL_ACCESS = "full_access"

@dataclass
class KeyPolicy:
    """API key policy and restrictions"""
    max_requests_per_hour: int = 1000
    max_requests_per_day: int = 10000
    allowed_endpoints: List[str] = field(default_factory=list)
    blocked_endpoints: List[str] = field(default_factory=list)
    allowed_ips: List[str] = field(default_factory=list)
    blocked_ips: List[str] = field(default_factory=list)
    rate_limit_burst: int = 100
    require_mfa: bool = False
    allow_renewal: bool = True
    renewable_days_before_expiry: int = 7

@dataclass
class AdvancedAPIKey:
    """Enhanced API key with comprehensive security features"""
    key_id: str
    user_id: str
```

```
    key_hash: str
    key_prefix: str
    key_suffix: str  # For additional security

    # Status and policy
    status: KeyStatus = KeyStatus.ACTIVE
    usage_type: KeyUsage = KeyUsage.READ_ONLY
    policy: KeyPolicy = field(default_factory=KeyPolicy)

    # Timestamps
    created_at: datetime = field(default_factory=datetime.utcnow)
    last_used: Optional[datetime] = None
    expires_at: Optional[datetime] = None
    renewed_at: Optional[datetime] = None

    # Usage tracking
    usage_count: int = 0
    hourly_usage: Dict[str, int] = field(default_factory=dict)
    daily_usage: Dict[str, int] = field(default_factory=dict)
    endpoint_usage: Dict[str, int] = field(default_factory=dict)

    # Security metadata
    created_by_ip: str = ""
    last_used_ip: str = ""
    failed_attempts: int = 0
    rotation_count: int = 0

    def is_valid(self) -> bool:
        """Check if key is currently valid"""
        if self.status != KeyStatus.ACTIVE:
            return False

        if self.expires_at and datetime.utcnow() > self.expires_at:
            return False

        return True

    def can_renew(self) -> bool:
        """Check if key can be renewed"""
        if not self.policy.allow_renewal:
            return False
```

```
        if not self.expires_at:
            return False

        days_until_expiry = (self.expires_at - datetime.utcnow()).days
        return days_until_expiry <=
self.policy.renewable_days_before_expiry

class APIKeyManager:
    """Advanced API key management system"""

    def __init__(self):
        self.logger = logging.getLogger(__name__)

        # Storage (in production, use secure database)
        self.keys: Dict[str, AdvancedAPIKey] = {}
        self.key_lookup: Dict[str, str] = {}  # Hash -> Key ID mapping

        # Usage tracking
        self.request_history: List[Dict] = []

        # Security settings
        self.security_config = {
            'max_failed_attempts': 5,
            'lockout_duration_minutes': 60,
            'default_expiry_days': 90,
            'key_rotation_interval_days': 30,
            'enable_key_suffix': True,
            'min_key_length': 32
        }

    async def create_api_key(self,
                             user_id: str,
                             usage_type: KeyUsage,
                             policy: Optional[KeyPolicy] = None,
                             expires_in_days: Optional[int] = None,
                             created_by_ip: str = "") -> Tuple[str,
AdvancedAPIKey]:
        """Create new API key with security policy"""

        # Generate secure key components
        raw_key = self._generate_secure_key()
        key_hash = hashlib.sha256(raw_key.encode()).hexdigest()
```

```
        key_prefix = raw_key[:8]
        key_suffix = raw_key[-4:] if
self.security_config['enable_key_suffix'] else ""

        # Create key ID
        key_id = secrets.token_urlsafe(16)

        # Create policy if not provided
        if policy is None:
            policy = KeyPolicy()

        # Set expiry
        expires_at = None
        if expires_in_days:
            expires_at = datetime.utcnow() +
timedelta(days=expires_in_days)
        else:
            expires_at = datetime.utcnow() +
timedelta(days=self.security_config['default_expiry_days'])

        # Create advanced API key
        api_key = AdvancedAPIKey(
            key_id=key_id,
            user_id=user_id,
            key_hash=key_hash,
            key_prefix=key_prefix,
            key_suffix=key_suffix,
            usage_type=usage_type,
            policy=policy,
            expires_at=expires_at,
            created_by_ip=created_by_ip
        )

        # Store key
        self.keys[key_id] = api_key
        self.key_lookup[key_hash] = key_id

        self.logger.info(f"Created API key {key_id} for user
{user_id}")

        # Return full key and key object
        full_key = f"{key_prefix}.{key_id}.{key_suffix if key_suffix
```

```
else secrets.token_urlsafe(8)}"
        return full_key, api_key

    def _generate_secure_key(self) -> str:
        """Generate cryptographically secure API key"""
        # Generate base key
        base_key =
secrets.token_urlsafe(self.security_config['min_key_length'])

        # Add checksum for integrity
        checksum = hashlib.sha256(base_key.encode()).hexdigest()[:8]

        return f"{base_key}.{checksum}"

    async def authenticate_api_key(self,
                                   provided_key: str,
                                   source_ip: str,
                                   user_agent: str) ->
Optional[AdvancedAPIKey]:
        """Authenticate API key with comprehensive checks"""

        try:
            # Parse provided key
            key_components = provided_key.split('.')
            if len(key_components) < 2:
                return None

            key_prefix = key_components[0]
            key_id = key_components[1]
            key_suffix = key_components[2] if len(key_components) > 2
else ""

            # Find key by ID
            api_key = self.keys.get(key_id)
            if not api_key:
                await self._log_authentication_failure(None, source_ip,
"key_not_found")
                return None

            # Verify key components
            if not self._verify_key_components(api_key, provided_key):
                api_key.failed_attempts += 1
```

```
                await self._log_authentication_failure(api_key.user_id,
source_ip, "invalid_key")

                # Check for brute force attempts
                if api_key.failed_attempts >=
self.security_config['max_failed_attempts']:
                    api_key.status = KeyStatus.SUSPENDED
                    await self._log_security_event(api_key,
"KEY_SUSPENDED", source_ip)

                return None

            # Check key validity
            if not api_key.is_valid():
                await self._log_authentication_failure(api_key.user_id,
source_ip, "invalid_status")
                return None

            # Check IP restrictions
            if not self._check_ip_restrictions(api_key, source_ip):
                await self._log_authentication_failure(api_key.user_id,
source_ip, "ip_restricted")
                return None

            # Reset failed attempts on success
            api_key.failed_attempts = 0

            # Update usage tracking
            await self._update_usage_tracking(api_key, source_ip,
user_agent)

            # Log successful authentication
            await self._log_security_event(api_key,
"KEY_AUTHENTICATED", source_ip)

            return api_key

        except Exception as e:
            self.logger.error(f"API key authentication error: {e}")
            return None

    def _verify_key_components(self, api_key: AdvancedAPIKey,
```

```
provided_key: str) -> bool:
        """Verify all components of API key"""

        try:
            # Parse provided key
            key_components = provided_key.split('.')
            if len(key_components) < 2:
                return False

            key_prefix = key_components[0]
            key_id = key_components[1]
            key_suffix = key_components[2] if len(key_components) > 2
else ""

            # Check prefix and suffix
            if key_prefix != api_key.key_prefix:
                return False

            if api_key.key_suffix and key_suffix != api_key.key_suffix:
                return False

            # Verify hash (reconstruct full key and hash it)
            reconstructed_key = f"{key_prefix}.{key_id}.{key_suffix}"
            reconstructed_hash =
hashlib.sha256(reconstructed_key.encode()).hexdigest()

            return reconstructed_hash == api_key.key_hash

        except Exception:
            return False

    def _check_ip_restrictions(self, api_key: AdvancedAPIKey,
source_ip: str) -> bool:
        """Check IP address restrictions"""

        policy = api_key.policy

        # Check blocked IPs first
        if source_ip in policy.blocked_ips:
            return False

        # Check allowed IPs (if any specified)
```

```
        if policy.allowed_ips and source_ip not in policy.allowed_ips:
            return False

        return True

    async def _update_usage_tracking(self,
                                     api_key: AdvancedAPIKey,
                                     source_ip: str,
                                     user_agent: str):
        """Update usage tracking for API key"""

        now = datetime.utcnow()

        # Update last used
        api_key.last_used = now
        api_key.last_used_ip = source_ip
        api_key.usage_count += 1

        # Update hourly usage
        hour_key = now.strftime("%Y-%m-%d-%H")
        api_key.hourly_usage[hour_key] =
api_key.hourly_usage.get(hour_key, 0) + 1

        # Update daily usage
        day_key = now.strftime("%Y-%m-%d")
        api_key.daily_usage[day_key] = api_key.daily_usage.get(day_key,
0) + 1

        # Record request
        self.request_history.append({
            'timestamp': now,
            'key_id': api_key.key_id,
            'user_id': api_key.user_id,
            'source_ip': source_ip,
            'user_agent': user_agent
        })

        # Clean old history (keep last 30 days)
        cutoff_date = now - timedelta(days=30)
        self.request_history = [
            req for req in self.request_history
            if req['timestamp'] > cutoff_date
```

```
        ]

    async def check_rate_limit(self, api_key: AdvancedAPIKey, endpoint:
str) -> bool:
        """Check rate limit for API key and endpoint"""

        policy = api_key.policy
        now = datetime.utcnow()

        # Check hourly limit
        hour_key = now.strftime("%Y-%m-%d-%H")
        hourly_usage = api_key.hourly_usage.get(hour_key, 0)

        if hourly_usage >= policy.max_requests_per_hour:
            await self._log_security_event(api_key,
"RATE_LIMIT_HOURLY", api_key.last_used_ip)
            return False

        # Check daily limit
        day_key = now.strftime("%Y-%m-%d")
        daily_usage = api_key.daily_usage.get(day_key, 0)

        if daily_usage >= policy.max_requests_per_day:
            await self._log_security_event(api_key, "RATE_LIMIT_DAILY",
api_key.last_used_ip)
            return False

        # Update endpoint usage
        api_key.endpoint_usage[endpoint] =
api_key.endpoint_usage.get(endpoint, 0) + 1

        return True

    async def rotate_api_key(self, key_id: str) -> Optional[Tuple[str,
AdvancedAPIKey]]:
        """Rotate API key for security"""

        api_key = self.keys.get(key_id)
        if not api_key:
            return None

        # Generate new key
```

```
        new_raw_key = self._generate_secure_key()
        new_key_hash = hashlib.sha256(new_raw_key.encode()).hexdigest()
        new_key_prefix = new_raw_key[:8]
        new_key_suffix = new_raw_key[-4:] if
self.security_config['enable_key_suffix'] else ""

        # Update key
        api_key.key_hash = new_key_hash
        api_key.key_prefix = new_key_prefix
        api_key.key_suffix = new_key_suffix
        api_key.renewed_at = datetime.utcnow()
        api_key.rotation_count += 1

        # Update lookup
        self.key_lookup[new_key_hash] = key_id

        await self._log_security_event(api_key, "KEY_ROTATED",
api_key.last_used_ip)

        # Return new key
        new_full_key = f"{new_key_prefix}.{key_id}.{new_key_suffix}"
        return new_full_key, api_key

    async def revoke_api_key(self, key_id: str, reason: str =
"manual_revoke") -> bool:
        """Revoke API key"""

        api_key = self.keys.get(key_id)
        if not api_key:
            return False

        api_key.status = KeyStatus.REVOKED

        await self._log_security_event(api_key, "KEY_REVOKED",
api_key.last_used_ip, {"reason": reason})

        return True

    async def suspend_api_key(self, key_id: str, reason: str =
"security_suspension") -> bool:
        """Suspend API key temporarily"""
```

```
        api_key = self.keys.get(key_id)
        if not api_key:
            return False

        api_key.status = KeyStatus.SUSPENDED

        await self._log_security_event(api_key, "KEY_SUSPENDED",
api_key.last_used_ip, {"reason": reason})

        return True

    async def reactivate_api_key(self, key_id: str) -> bool:
        """Reactivate suspended or expired key"""

        api_key = self.keys.get(key_id)
        if not api_key:
            return False

        # Check if key can be reactivated
        if api_key.status == KeyStatus.REVOKED:
            return False  # Permanently revoked keys cannot be
reactivated

        api_key.status = KeyStatus.ACTIVE
        api_key.failed_attempts = 0

        await self._log_security_event(api_key, "KEY_REACTIVATED",
api_key.last_used_ip)

        return True

    def get_key_statistics(self, key_id: str) -> Optional[Dict[str,
Any]]:
        """Get comprehensive statistics for API key"""

        api_key = self.keys.get(key_id)
        if not api_key:
            return None

        # Calculate usage statistics
        total_usage = api_key.usage_count
        days_since_created = (datetime.utcnow() -
```

```
api_key.created_at).days
        days_since_last_used = (
            (datetime.utcnow() - api_key.last_used).days
            if api_key.last_used else None
        )

        # Average usage per day
        avg_daily_usage = total_usage / max(days_since_created, 1)

        # Top endpoints
        top_endpoints = sorted(
            api_key.endpoint_usage.items(),
            key=lambda x: x[1],
            reverse=True
        )[:5]

        return {
            "key_id": api_key.key_id,
            "user_id": api_key.user_id,
            "status": api_key.status.value,
            "usage_type": api_key.usage_type.value,
            "created_at": api_key.created_at.isoformat(),
            "expires_at": api_key.expires_at.isoformat() if
api_key.expires_at else None,
            "last_used": api_key.last_used.isoformat() if
api_key.last_used else None,
            "total_usage": total_usage,
            "avg_daily_usage": avg_daily_usage,
            "days_since_created": days_since_created,
            "days_since_last_used": days_since_last_used,
            "rotation_count": api_key.rotation_count,
            "failed_attempts": api_key.failed_attempts,
            "top_endpoints": top_endpoints,
            "policy": {
                "max_requests_per_hour":
api_key.policy.max_requests_per_hour,
                "max_requests_per_day":
api_key.policy.max_requests_per_day,
                "allowed_ips": api_key.policy.allowed_ips,
                "blocked_ips": api_key.policy.blocked_ips,
                "require_mfa": api_key.policy.require_mfa
            }
```

```
        }

    async def get_keys_expiring_soon(self, days_ahead: int = 7) ->
List[AdvancedAPIKey]:
        """Get keys expiring within specified days"""

        cutoff_date = datetime.utcnow() + timedelta(days=days_ahead)

        expiring_keys = [
            key for key in self.keys.values()
            if (key.status == KeyStatus.ACTIVE and
                key.expires_at and
                key.expires_at <= cutoff_date)
        ]

        return expiring_keys

    async def cleanup_expired_keys(self) -> int:
        """Clean up expired keys"""

        now = datetime.utcnow()
        expired_keys = [
            key_id for key_id, key in self.keys.items()
            if key.expires_at and key.expires_at < now and key.status
== KeyStatus.ACTIVE
        ]

        for key_id in expired_keys:
            self.keys[key_id].status = KeyStatus.EXPIRED
            await self._log_security_event(
                self.keys[key_id], "KEY_EXPIRED",
self.keys[key_id].last_used_ip
            )

        return len(expired_keys)

    async def _log_authentication_failure(self,
                                          user_id: Optional[str],
                                          source_ip: str,
                                          reason: str):
        """Log authentication failure"""
```

```
        self.logger.warning(f"API key authentication failed:
{reason} - IP: {source_ip} - User: {user_id}")

    async def _log_security_event(self,
                                  api_key: AdvancedAPIKey,
                                  event_type: str,
                                  source_ip: str,
                                  additional_details: Optional[Dict] =
None):
        """Log security event"""

        details = {
            "key_id": api_key.key_id,
            "user_id": api_key.user_id,
            "key_prefix": api_key.key_prefix
        }

        if additional_details:
            details.update(additional_details)

        self.logger.info(f"Security event: {event_type} -
{json.dumps(details)}")
```

Summary

This comprehensive module on security and authentication provides essential skills for building secure MEV systems. Key achievements:

✅ What You Learned:

- **Comprehensive security architecture** with layered defense and zero-trust principles
- **Advanced API key management** with rotation, policies, and comprehensive tracking
- **OAuth 2.0 and OpenID Connect** implementation for user authentication
- **Multi-factor authentication** with various second factor methods
- **Secure communication channels** with TLS/mTLS and certificate management
- **Access control systems** with RBAC and fine-grained permissions

🚀 Production Implementation:

- **Enterprise-grade security** with audit logging and compliance frameworks

- **Automated security monitoring** with threat detection and response
- **Key management infrastructure** with HSM integration and rotation policies
- **Identity and access management** with SSO and federated authentication
- **Data protection systems** with encryption at rest and in transit

💡 Best Practices:

- **Security by design** with threat modeling and secure defaults
- **Principle of least privilege** for all access controls
- **Defense in depth** with multiple security layers
- **Continuous security monitoring** with real-time threat detection
- **Incident response planning** with proper escalation and recovery procedures

The security and authentication skills from this module enable building MEV systems with enterprise-grade security, ensuring protection of sensitive trading strategies, user data, and operational infrastructure while maintaining regulatory compliance and audit requirements.

Course Completion Summary

You have now completed all 6 modules of the **API Integration** course:

1. ✅ **Module 1: Blockchain API Integration** - Master Ethereum, BSC, and Layer 2 API clients
2. ✅ **Module 2: DeFi Protocol APIs** - Build DEX, lending, and yield farming integrations
3. ✅ **Module 3: Real-time Data Streaming** - Implement WebSocket connections and event processing
4. ✅ **Module 4: Infrastructure Automation** - Master Docker, Kubernetes, and cloud deployment
5. ✅ **Module 5: Monitoring & Analytics** - Build comprehensive observability with Prometheus and Grafana
6. ✅ **Module 6: Security & Authentication** - Implement secure API management and compliance

Total Course Stats:
- **Duration:** 14 hours (840 minutes)
- **Content:** 6 comprehensive modules with practical code examples
- **Skills:** Production-ready MEV system integration capabilities
- **Certificate:** Available upon completion

You now have the expertise to build, deploy, and operate sophisticated MEV systems with enterprise-grade API integration, real-time data processing, infrastructure automation, monitoring, and security!