```python
import base64
import hashlib
import json
import os
import secrets
import threading
import time
from dataclasses import dataclass
from datetime import datetime, timedelta
from functools import wraps
from typing import Any, List, Optional

from cryptography.exceptions import InvalidKey, InvalidSignature
from cryptography.fernet import Fernet, MultiFernet
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from loguru import logger


@dataclass
class EncryptionKey:
    """Represents an encryption key with metadata"""

    key_id: str
    key: bytes
```

```python
    created_at: datetime
    expires_at: datetime
    is_primary: bool = False


class KeyRotationPolicy:
    """Manages key rotation policies and schedules"""

    def __init__(
        self,
        rotation_interval: timedelta = timedelta(days=30),
        key_overlap_period: timedelta = timedelta(days=2),
    ):
        self.rotation_interval = rotation_interval
        self.key_overlap_period = key_overlap_period

    def should_rotate(self, key: EncryptionKey) -> bool:
        """Check if a key should be rotated based on policy"""
        time_until_expiry = key.expires_at - datetime.now()
        return time_until_expiry <= self.key_overlap_period

    def get_next_rotation_time(self, key: EncryptionKey) -> datetime:
        """Calculate the next rotation time for a key"""
        return (
            key.created_at
            + self.rotation_interval
```

```
            - self.key_overlap_period
        )


class SecureDataHandler:
    """Production-grade secure data handler with key rotation and versioning"""

    VERSION = "2.0"
    KEY_ITERATIONS = 200000  # Increased from 100000
    SALT_LENGTH = 32  # 256 bits

    def __init__(
        self,
        master_key: str = os.getenv("MASTER_KEY"),
        key_storage_path: Optional[str] = None,
        rotation_policy: Optional[KeyRotationPolicy] = None,
        auto_rotate: bool = True,
    ):
        """
        Initialize the secure data handler with enhanced security features.

        Args:
            master_key: Master encryption key
            key_storage_path: Path to store encrypted key material
            rotation_policy: Key rotation policy configuration
            auto_rotate: Whether to automatically rotate keys
```

```
    """
    self.master_key_hash = hashlib.sha256(
        master_key.encode()
    ).hexdigest()
    self.key_storage_path = key_storage_path or os.path.join(
        os.getcwd(), ".secure_keys"
    )
    self.rotation_policy = rotation_policy or KeyRotationPolicy()
    self.auto_rotate = auto_rotate

    # Thread-safe key management
    self._keys_lock = threading.RLock()
    self._active_keys: List[EncryptionKey] = []
    self._primary_key: Optional[EncryptionKey] = None

    # Initialize key storage
    os.makedirs(self.key_storage_path, exist_ok=True)

    # Setup initial keys if none exist
    self._initialize_keys(master_key)

    # Start key rotation monitor if auto-rotate is enabled
    if auto_rotate:
        self._start_key_rotation_monitor()

def _derive_key(self, master_key: str, salt: bytes) -> bytes:
```

```python
    """Derive a key using PBKDF2 with enhanced security"""
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=self.KEY_ITERATIONS,
        backend=default_backend(),
    )
    return base64.urlsafe_b64encode(
        kdf.derive(master_key.encode())
    )

def _initialize_keys(self, master_key: str) -> None:
    """Initialize encryption keys"""
    with self._keys_lock:
        if not self._load_existing_keys():
            # Generate initial key
            self._generate_new_key(master_key, is_primary=True)

def _generate_new_key(
    self, master_key: str, is_primary: bool = False
) -> EncryptionKey:
    """Generate a new encryption key"""
    key_id = secrets.token_hex(16)
    salt = os.urandom(self.SALT_LENGTH)
    key = self._derive_key(master_key, salt)
```

```
    encryption_key = EncryptionKey(
        key_id=key_id,
        key=key,
        created_at=datetime.now(),
        expires_at=datetime.now()
        + self.rotation_policy.rotation_interval,
        is_primary=is_primary,
    )

    # Save key securely
    self._save_key(encryption_key, salt)

    with self._keys_lock:
        self._active_keys.append(encryption_key)
        if is_primary:
            self._primary_key = encryption_key

    return encryption_key

def _save_key(self, key: EncryptionKey, salt: bytes) -> None:
    """Save key material securely"""
    key_data = {
        "key_id": key.key_id,
        "salt": base64.b64encode(salt).decode(),
        "created_at": key.created_at.isoformat(),
```

```
        "expires_at": key.expires_at.isoformat(),
        "is_primary": key.is_primary,
    }

    key_path = os.path.join(
        self.key_storage_path, f"{key.key_id}.key"
    )
    with open(key_path, "w") as f:
        json.dump(key_data, f)

def _load_existing_keys(self) -> bool:
    """Load existing keys from storage"""
    key_files = [
        f
        for f in os.listdir(self.key_storage_path)
        if f.endswith(".key")
    ]
    if not key_files:
        return False

    for key_file in key_files:
        try:
            with open(
                os.path.join(self.key_storage_path, key_file)
            ) as f:
                key_data = json.load(f)
```

```
                salt = base64.b64decode(key_data["salt"])
                key = self._derive_key(self.master_key_hash, salt)

                encryption_key = EncryptionKey(
                    key_id=key_data["key_id"],
                    key=key,
                    created_at=datetime.fromisoformat(
                        key_data["created_at"]
                    ),
                    expires_at=datetime.fromisoformat(
                        key_data["expires_at"]
                    ),
                    is_primary=key_data["is_primary"],
                )

                self._active_keys.append(encryption_key)
                if encryption_key.is_primary:
                    self._primary_key = encryption_key

            except Exception as e:
                logger.error(f"Error loading key {key_file}: {e}")
                continue

        return bool(self._active_keys)
```

```python
def _start_key_rotation_monitor(self) -> None:
    """Start background thread for key rotation"""

    def monitor_keys():
        while True:
            try:
                self._check_and_rotate_keys()
            except Exception as e:
                logger.error(
                    f"Error in key rotation monitor: {e}"
                )
            time.sleep(3600)  # Check every hour

    threading.Thread(target=monitor_keys, daemon=True).start()

def _check_and_rotate_keys(self) -> None:
    """Check and rotate keys based on policy"""
    with self._keys_lock:
        if (
            not self._primary_key
            or self.rotation_policy.should_rotate(
                self._primary_key
            )
        ):
            # Generate new primary key
            new_primary = self._generate_new_key(
```

```
                self.master_key_hash, is_primary=True
            )
            old_primary = self._primary_key

            # Update primary key
            self._primary_key = new_primary
            if old_primary:
                old_primary.is_primary = False

            # Remove expired keys
            self._clean_expired_keys()

    def _clean_expired_keys(self) -> None:
        """Remove expired keys"""
        now = datetime.now()
        with self._keys_lock:
            self._active_keys = [
                k
                for k in self._active_keys
                if k.expires_at > now or k.is_primary
            ]

    def _get_fernet(self) -> MultiFernet:
        """Get MultiFernet instance with all active keys"""
        with self._keys_lock:
            if not self._active_keys:
```

```
            raise ValueError(
                "No active encryption keys available"
            )
        fernets = [Fernet(key.key) for key in self._active_keys]
        return MultiFernet(fernets)

    def encrypt_data(self, data: Any) -> str:
        """
        Encrypt data with version control and integrity checking.

        Args:
            data: Data to encrypt

        Returns:
            str: Versioned and encrypted data in base64 format
        """
        try:
            # Add metadata
            payload = {
                "version": self.VERSION,
                "timestamp": datetime.now().isoformat(),
                "data": data,
            }

            json_data = json.dumps(payload)
```

```
            # Calculate checksum
            checksum = hashlib.sha256(json_data.encode()).hexdigest()

            # Add checksum to payload
            payload["checksum"] = checksum

            # Encrypt with primary key
            fernet = self._get_fernet()
            encrypted_data = fernet.encrypt(
                json.dumps(payload).encode()
            )

            return base64.urlsafe_b64encode(encrypted_data).decode()

        except Exception as e:
            logger.error(f"Encryption error: {e}")
            raise EncryptionError(f"Failed to encrypt data: {str(e)}")

    def decrypt_data(self, encrypted_data: str) -> Any:
        """
        Decrypt data with version handling and integrity verification.

        Args:
            encrypted_data: Encrypted data in base64 format

        Returns:
```

```python
        Any: Decrypted data
    """
    try:
        # Decrypt data
        encrypted_bytes = base64.urlsafe_b64decode(
            encrypted_data.encode()
        )
        fernet = self._get_fernet()
        decrypted_data = fernet.decrypt(encrypted_bytes)

        # Parse payload
        payload = json.loads(decrypted_data)

        # Verify version
        if payload["version"] != self.VERSION:
            logger.warning(
                f"Decrypting data from version {payload['version']}"
            )

        # Extract and verify checksum
        stored_checksum = payload.pop("checksum", None)
        if stored_checksum:
            # Reconstruct original payload for checksum verification
            verification_payload = {
                "version": payload["version"],
                "timestamp": payload["timestamp"],
```

```python
                "data": payload["data"],
            }
            calculated_checksum = hashlib.sha256(
                json.dumps(verification_payload).encode()
            ).hexdigest()

            if stored_checksum != calculated_checksum:
                raise IntegrityError(
                    "Data integrity check failed"
                )

        return payload["data"]

    except (InvalidSignature, InvalidKey) as e:
        logger.error(f"Decryption error: {e}")
        raise DecryptionError(f"Failed to decrypt data: {str(e)}")
    except json.JSONDecodeError as e:
        logger.error(f"Invalid data format: {e}")
        raise DecryptionError("Invalid encrypted data format")
    except Exception as e:
        logger.error(f"Unexpected error during decryption: {e}")
        raise


class EncryptionError(Exception):
    """Raised when encryption fails"""
```

```
    pass


class DecryptionError(Exception):
    """Raised when decryption fails"""

    pass


class IntegrityError(Exception):
    """Raised when data integrity check fails"""

    pass


# Decorator for automatic encryption/decryption
def secure_data(encrypt: bool = True):
    def decorator(func):
        @wraps(func)
        def wrapper(self, *args, **kwargs):
            result = func(self, *args, **kwargs)
            if encrypt and isinstance(result, (dict, list, str)):
                return self.secure_handler.encrypt_data(result)
            return result
```

```
    return wrapper

return decorator
```