Техническое описание PAM-системы

Введение

Данный документ содержит подробное техническое описание системы управления привилегированным доступом (Privileged Access Management - PAM), разработанной в рамках дипломной работы. PAM-система представляет собой комплексное решение для обеспечения безопасности критически важных ресурсов организации через централизованное управление привилегированными учетными данными, мониторинг сессий и контроль доступа.

Система реализована с использованием современных технологий и следует принципам безопасности по умолчанию, обеспечивая высокий уровень защиты при сохранении удобства использования для конечных пользователей и администраторов.

Архитектура системы

Общая архитектура

PAM-система построена по трехуровневой архитектуре:

1. **Уровень представления (Presentation Layer)** - веб-интерфейс пользователя, реализованный с использованием HTML5, CSS3, JavaScript и Bootstrap 5
2. **Уровень бизнес-логики (Business Logic Layer)** - серверная часть на Flask с REST API для обработки запросов
3. **Уровень данных (Data Layer)** - база данных SQLite/PostgreSQL для хранения всей информации системы

Принципы проектирования

Система спроектирована с учетом следующих принципов:

- **Модульность**: каждый компонент системы выделен в отдельный модуль с четко определенными интерфейсами
- **Масштабируемость**: архитектура позволяет легко добавлять новые функции и компоненты
- **Безопасность**: все данные шифруются, пароли хешируются, доступ контролируется
- **Аудитируемость**: все действия пользователей логируются для последующего анализа
- **Отказоустойчивость**: система продолжает работать даже при частичных сбоях

Структура проекта

```
pam_system/
├── src/
│   ├── main.py                 # Главный файл приложения
│   ├── models/                 # Модели данных
│   │   ├── privileged_user.py  # Модель пользователей
│   │   ├── credential.py       # Модель учетных данных
│   │   ├── session.py          # Модель сессий
│   │   ├── audit.py           # Модель аудита
│   │   └── policy.py          # Модель политик
│   ├── routes/                 # API маршруты
│   │   ├── auth.py            # Аутентификация
│   │   ├── users.py           # Управление пользователями
│   │   ├── credentials.py     # Управление учетными данными
│   │   ├── sessions.py        # Управление сессиями
│   │   ├── audit.py           # Аудит и отчетность
│   │   ├── policies.py        # Политики доступа
│   │   └── dashboard.py       # Дашборд
│   └── static/
│       └── index.html         # Веб-интерфейс
├── requirements.txt           # Зависимости Python
└── README.md                 # Документация
```

Модели данных

1. Модель пользователей (PrivilegedUser)

Модель `PrivilegedUser` представляет пользователей системы и содержится в файле `src/models/privileged_user.py`.

Структура модели

```
class PrivilegedUser(db.Model):
    __tablename__ = 'privileged_users'

    # Основные поля
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False,
index=True)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password_hash = db.Column(db.String(255), nullable=False)
    full_name = db.Column(db.String(200), nullable=False)
    department = db.Column(db.String(100))

    # Роль и статус
    role = db.Column(db.Enum(UserRole), nullable=False, default=UserRole.USER)
    status = db.Column(db.Enum(UserStatus), nullable=False,
default=UserStatus.ACTIVE)

    # Временные ограничения
    access_start_time = db.Column(db.Time)
    access_end_time = db.Column(db.Time)

    # Метаданные
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow,
onupdate=datetime.utcnow)
    last_login = db.Column(db.DateTime)
    failed_login_attempts = db.Column(db.Integer, default=0)
```

Роли пользователей

Система поддерживает четыре роли пользователей:

- **ADMIN** - полный доступ ко всем функциям системы
- **OPERATOR** - управление учетными данными и сессиями
- **AUDITOR** - доступ только к журналам аудита и отчетам
- **USER** - базовый доступ к назначенным ресурсам

Статусы пользователей

- **ACTIVE** - пользователь активен и может входить в систему
- **INACTIVE** - пользователь временно деактивирован
- **SUSPENDED** - пользователь приостановлен из-за нарушений
- **LOCKED** - пользователь заблокирован после превышения лимита неудачных попыток входа

Ключевые методы

set_password(password) - устанавливает хеш пароля с использованием Werkzeug

```
def set_password(self, password):
    """Устанавливает хэш пароля"""
    self.password_hash = generate_password_hash(password)
```

check_password(password) - проверяет соответствие пароля хешу

```
def check_password(self, password):
    """Проверяет пароль"""
    return check_password_hash(self.password_hash, password)
```

is_active() - проверяет активность пользователя

```
def is_active(self):
    """Проверяет, активен ли пользователь"""
    return self.status == UserStatus.ACTIVE
```

can_access_now() - проверяет временные ограничения доступа

```
def can_access_now(self):
    """Проверяет, может ли пользователь получить доступ в текущее время"""
    if not self.is_active():
        return False

    if self.access_start_time and self.access_end_time:
        current_time = datetime.now().time()
        return self.access_start_time <= current_time <= self.access_end_time

    return True
```

2. Модель учетных данных (PrivilegedCredential)

Модель `PrivilegedCredential` управляет привилегированными учетными данными и находится в файле `src/models/credential.py`.

Структура модели

```
class PrivilegedCredential(db.Model):
    __tablename__ = 'privileged_credentials'

    # Основные поля
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(200), nullable=False)
    description = db.Column(db.Text)

    # Тип и статус
    credential_type = db.Column(db.Enum(CredentialType), nullable=False)
    status = db.Column(db.Enum(CredentialStatus), nullable=False,
default=CredentialStatus.ACTIVE)

    # Целевая система
    target_system = db.Column(db.String(200), nullable=False)
    target_host = db.Column(db.String(255), nullable=False)
    target_port = db.Column(db.Integer)
    target_username = db.Column(db.String(100), nullable=False)

    # Зашифрованные данные
    encrypted_password = db.Column(db.Text)
    encrypted_private_key = db.Column(db.Text)
    encryption_key_id = db.Column(db.String(100))

    # Ротация паролей
    rotation_enabled = db.Column(db.Boolean, default=False)
    rotation_interval_days = db.Column(db.Integer, default=90)
    last_rotation = db.Column(db.DateTime)
    next_rotation = db.Column(db.DateTime)
```

Типы учетных данных

- **PASSWORD** - обычные пароли
- **SSH_KEY** - SSH ключи для удаленного доступа
- **API_KEY** - ключи для API доступа
- **DATABASE** - учетные данные для баз данных
- **CERTIFICATE** - цифровые сертификаты

Статусы учетных данных

- **ACTIVE** - учетные данные активны и доступны для использования

- **INACTIVE** - временно недоступны
- **EXPIRED** - срок действия истек
- **COMPROMISED** - скомпрометированы и требуют замены

Ключевые методы

encrypt_credential(credential, key) - шифрует учетные данные

```
def encrypt_credential(self, credential, key):
    """Шифрует учетные данные"""
    from cryptography.fernet import Fernet
    fernet = Fernet(key)
    encrypted_data = fernet.encrypt(credential.encode())
    self.encrypted_password = encrypted_data.decode()
    self.encryption_key_id = hashlib.sha256(key).hexdigest()[:16]
```

decrypt_credential(key) - расшифровывает учетные данные

```
def decrypt_credential(self, key):
    """Расшифровывает учетные данные"""
    if not self.encrypted_password:
        return None

    from cryptography.fernet import Fernet
    fernet = Fernet(key)
    decrypted_data = fernet.decrypt(self.encrypted_password.encode())
    return decrypted_data.decode()
```

schedule_rotation() - планирует ротацию учетных данных

```
def schedule_rotation(self):
    """Планирует следующую ротацию"""
    if self.rotation_enabled and self.rotation_interval_days:
        self.next_rotation = datetime.utcnow() +
timedelta(days=self.rotation_interval_days)
```

3. Модель сессий (PrivilegedSession)

Модель `PrivilegedSession` отслеживает привилегированные сессии пользователей и содержится в файле `src/models/session.py`.

Структура модели

```
class PrivilegedSession(db.Model):
    __tablename__ = 'privileged_sessions'

    # Основные поля
    id = db.Column(db.Integer, primary_key=True)
    session_token = db.Column(db.String(255), unique=True, nullable=False,
index=True)

    # Пользователь и система
    user_id = db.Column(db.Integer, db.ForeignKey('privileged_users.id'),
nullable=False)
    target_system = db.Column(db.String(200), nullable=False)
    target_host = db.Column(db.String(255), nullable=False)
    target_port = db.Column(db.Integer)

    # Тип и статус
    session_type = db.Column(db.Enum(SessionType), nullable=False)
    status = db.Column(db.Enum(SessionStatus), nullable=False,
default=SessionStatus.ACTIVE)

    # Временные рамки
    start_time = db.Column(db.DateTime, default=datetime.utcnow)
    end_time = db.Column(db.DateTime)
    max_duration_minutes = db.Column(db.Integer, default=480)
    idle_timeout_minutes = db.Column(db.Integer, default=30)
    last_activity = db.Column(db.DateTime, default=datetime.utcnow)

    # Контекст доступа
    source_ip = db.Column(db.String(45))
    user_agent = db.Column(db.String(500))
    justification = db.Column(db.Text)
    approved_by = db.Column(db.Integer, db.ForeignKey('privileged_users.id'))

    # Мониторинг
    recording_enabled = db.Column(db.Boolean, default=True)
    recording_path = db.Column(db.String(500))
    commands_logged = db.Column(db.Integer, default=0)
```

Типы сессий

- **SSH** - SSH подключения к серверам
- **RDP** - удаленный рабочий стол
- **WEB** - веб-приложения
- **DATABASE** - подключения к базам данных
- **API** - API вызовы

Статусы сессий

- **ACTIVE** - сессия активна

- **TERMINATED** - сессия завершена
- **EXPIRED** - сессия истекла по времени
- **SUSPENDED** - сессия приостановлена

Ключевые методы

is_expired() - проверяет истечение сессии

```
def is_expired(self):
    """Проверяет, истекла ли сессия"""
    if self.status in [SessionStatus.EXPIRED, SessionStatus.TERMINATED]:
        return True

    # Проверка максимальной продолжительности
    if self.max_duration_minutes:
        max_end_time = self.start_time +
timedelta(minutes=self.max_duration_minutes)
        if datetime.utcnow() > max_end_time:
            return True

    # Проверка таймаута бездействия
    if self.idle_timeout_minutes and self.last_activity:
        idle_end_time = self.last_activity +
timedelta(minutes=self.idle_timeout_minutes)
        if datetime.utcnow() > idle_end_time:
            return True

    return False
```

terminate(reason) - завершает сессию

```
def terminate(self, reason="Manual termination"):
    """Завершает сессию"""
    self.status = SessionStatus.TERMINATED
    self.end_time = datetime.utcnow()

    # Логируем завершение
    activity = SessionActivity(
        session_id=self.id,
        activity_type="session_terminated",
        description=reason,
        timestamp=datetime.utcnow()
    )
    db.session.add(activity)
```

4. Модель аудита (AuditLog)

Модель `AuditLog` обеспечивает полное логирование всех действий в системе и находится в файле `src/models/audit.py`.

Структура модели

```
class AuditLog(db.Model):
    __tablename__ = 'audit_logs'

    # Основные поля
    id = db.Column(db.Integer, primary_key=True)
    event_type = db.Column(db.Enum(AuditEventType), nullable=False, index=True)
    severity = db.Column(db.Enum(AuditSeverity), nullable=False,
default=AuditSeverity.INFO)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow, index=True)

    # Пользователь и сессия
    user_id = db.Column(db.Integer, db.ForeignKey('privileged_users.id'),
index=True)
    session_id = db.Column(db.Integer, db.ForeignKey('privileged_sessions.id'))

    # Детали события
    title = db.Column(db.String(200), nullable=False)
    description = db.Column(db.Text)

    # Контекст
    source_ip = db.Column(db.String(45), index=True)
    user_agent = db.Column(db.String(500))
    target_system = db.Column(db.String(200))
    target_resource = db.Column(db.String(500))

    # Результат
    success = db.Column(db.Boolean, default=True)
    error_message = db.Column(db.Text)

    # Дополнительные данные
    event_metadata = db.Column(db.Text)  # JSON данные
    risk_score = db.Column(db.Integer, default=0)  # 0-100

    # Корреляция
    correlation_id = db.Column(db.String(100), index=True)
    parent_event_id = db.Column(db.Integer, db.ForeignKey('audit_logs.id'))
```

Типы событий аудита

- **LOGIN** - вход в систему
- **LOGOUT** - выход из системы
- **CREDENTIAL_ACCESS** - доступ к учетным данным
- **CREDENTIAL_ROTATION** - ротация учетных данных
- **SESSION_START** - начало привилегированной сессии
- **SESSION_END** - завершение сессии
- **POLICY_VIOLATION** - нарушение политики
- **SYSTEM_START** - запуск системы

- **CONFIGURATION_CHANGE** - изменение конфигурации

Уровни серьезности

- **INFO** - информационные события
- **LOW** - события низкой важности
- **MEDIUM** - события средней важности
- **HIGH** - важные события безопасности
- **CRITICAL** - критические события безопасности

5. Модель политик (AccessPolicy)

Модель `AccessPolicy` определяет правила доступа к ресурсам и содержится в файле `src/models/policy.py`.

Структура модели

```
class AccessPolicy(db.Model):
    __tablename__ = 'access_policies'

    # Основные поля
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(200), nullable=False)
    description = db.Column(db.Text)

    # Тип и статус
    policy_type = db.Column(db.Enum(PolicyType), nullable=False)
    status = db.Column(db.Enum(PolicyStatus), nullable=False,
default=PolicyStatus.ACTIVE)
    priority = db.Column(db.Integer, default=100)
    action = db.Column(db.Enum(PolicyAction), nullable=False)

    # Условия применения
    target_users = db.Column(db.Text)  # JSON список пользователей
    target_roles = db.Column(db.Text)  # JSON список ролей
    target_systems = db.Column(db.Text)  # JSON список систем
    source_ip_ranges = db.Column(db.Text)  # JSON список IP диапазонов

    # Временные ограничения
    time_start = db.Column(db.Time)
    time_end = db.Column(db.Time)
    days_of_week = db.Column(db.String(20))  # "1,2,3,4,5" для пн-пт

    # Метаданные
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    created_by = db.Column(db.Integer, db.ForeignKey('privileged_users.id'))
    last_modified = db.Column(db.DateTime, default=datetime.utcnow)
```

Типы политик

- **TIME_BASED** - ограничения по времени
- **IP_BASED** - ограничения по IP-адресам
- **ROLE_BASED** - ограничения по ролям
- **APPROVAL_REQUIRED** - требование утверждения

Действия политик

- **ALLOW** - разрешить доступ
- **DENY** - запретить доступ
- **REQUIRE_APPROVAL** - требовать утверждения

API маршруты

1. Аутентификация (auth.py)

Модуль аутентификации обеспечивает безопасный вход и выход пользователей из системы.

POST /api/auth/login

Выполняет аутентификацию пользователя в системе.

Параметры запроса:

```
{
    "username": "admin",
    "password": "admin123",
    "remember_me": false
}
```

Ответ при успехе:

```
{
    "success": true,
    "message": "Успешная аутентификация",
    "user": {
        "id": 1,
        "username": "admin",
        "role": "admin",
        "full_name": "System Administrator"
    },
    "session_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."
}
```

Логика работы: 1. Проверка существования пользователя 2. Валидация пароля 3. Проверка статуса пользователя (активен/заблокирован) 4. Проверка временных ограничений доступа 5. Генерация сессионного токена 6. Логирование события входа 7. Сброс счетчика неудачных попыток

Код реализации:

```python
@auth_bp.route('/login', methods=['POST'])
def login():
    try:
        data = request.get_json()
        username = data.get('username')
        password = data.get('password')

        if not username or not password:
            return jsonify({'success': False, 'message': 'Необходимо указать имя пользователя и пароль'}), 400

        # Поиск пользователя
        user = PrivilegedUser.query.filter_by(username=username).first()

        if not user or not user.check_password(password):
            # Логируем неудачную попытку
            audit_log = AuditLog(
                event_type=AuditEventType.LOGIN,
                title='Неудачная попытка входа',
                description=f'Неверные учетные данные для пользователя {username}',
                source_ip=request.remote_addr,
                user_agent=request.headers.get('User-Agent'),
                success=False,
                severity=AuditSeverity.MEDIUM
            )
            db.session.add(audit_log)

            if user:
                user.increment_failed_login()

            db.session.commit()
            return jsonify({'success': False, 'message': 'Неверные учетные данные'}), 401

        # Проверка статуса пользователя
        if not user.is_active():
            audit_log = AuditLog(
                event_type=AuditEventType.LOGIN,
                title='Попытка входа заблокированного пользователя',
                description=f'Пользователь {username} заблокирован',
                user_id=user.id,
                source_ip=request.remote_addr,
                success=False,
                severity=AuditSeverity.HIGH
            )
            db.session.add(audit_log)
            db.session.commit()
            return jsonify({'success': False, 'message': 'Учетная запись заблокирована'}), 403

        # Проверка временных ограничений
        if not user.can_access_now():
            return jsonify({'success': False, 'message': 'Доступ запрещен в текущее время'}), 403

        # Успешная аутентификация
        user.last_login = datetime.utcnow()
        user.reset_failed_login()

        # Генерация токена сессии
```

```
        session_token = secrets.token_urlsafe(32)
        session['user_id'] = user.id
        session['session_token'] = session_token
        session['login_time'] = datetime.utcnow().isoformat()

        # Логирование успешного входа
        audit_log = AuditLog(
            event_type=AuditEventType.LOGIN,
            title='Успешный вход в систему',
            description=f'Пользователь {username} успешно аутентифицирован',
            user_id=user.id,
            source_ip=request.remote_addr,
            user_agent=request.headers.get('User-Agent'),
            success=True,
            severity=AuditSeverity.INFO
        )
        db.session.add(audit_log)
        db.session.commit()

        return jsonify({
            'success': True,
            'message': 'Успешная аутентификация',
            'user': user.to_dict(),
            'session_token': session_token
        })

    except Exception as e:
        return jsonify({'success': False, 'message': f'Ошибка сервера:
{str(e)}'}), 500
```

POST /api/auth/logout

Завершает сессию пользователя.

Ответ:

```
{
    "success": true,
    "message": "Выход выполнен успешно"
}
```

GET /api/auth/check-session

Проверяет валидность текущей сессии.

Ответ:

```
{
    "authenticated": true,
    "user": {
        "id": 1,
        "username": "admin",
        "role": "admin"
    }
}
```

2. Управление пользователями (users.py)

Модуль управления пользователями предоставляет CRUD операции для работы с учетными записями.

GET /api/users

Получает список всех пользователей системы.

Параметры запроса (опциональные): - `page` - номер страницы (по умолчанию 1) - `per_page` - количество записей на странице (по умолчанию 20) - `role` - фильтр по роли - `status` - фильтр по статусу

Ответ:

```
{
    "success": true,
    "users": [
        {
            "id": 1,
            "username": "admin",
            "email": "admin@example.com",
            "full_name": "System Administrator",
            "role": "admin",
            "status": "active",
            "created_at": "2025-01-01T00:00:00",
            "last_login": "2025-01-01T12:00:00"
        }
    ],
    "pagination": {
        "page": 1,
        "per_page": 20,
        "total": 1,
        "pages": 1
    }
}
```

POST /api/users

Создает нового пользователя.

Параметры запроса:

```
{
    "username": "newuser",
    "email": "newuser@example.com",
    "password": "securepassword",
    "full_name": "New User",
    "department": "IT",
    "role": "user",
    "access_start_time": "09:00",
    "access_end_time": "18:00"
}
```

Логика создания пользователя:

```python
@users_bp.route('', methods=['POST'])
@require_auth
@require_role(['admin'])
def create_user():
    try:
        data = request.get_json()

        # Валидация обязательных полей
        required_fields = ['username', 'email', 'password', 'full_name']
        for field in required_fields:
            if not data.get(field):
                return jsonify({'success': False, 'message': f'Поле {field}
обязательно'}), 400

        # Проверка уникальности
        if PrivilegedUser.query.filter_by(username=data['username']).first():
            return jsonify({'success': False, 'message': 'Пользователь с таким
именем уже существует'}), 409

        if PrivilegedUser.query.filter_by(email=data['email']).first():
            return jsonify({'success': False, 'message': 'Пользователь с таким
email уже существует'}), 409

        # Создание пользователя
        user = PrivilegedUser(
            username=data['username'],
            email=data['email'],
            full_name=data['full_name'],
            department=data.get('department'),
            role=UserRole(data.get('role', 'user')),
            status=UserStatus.ACTIVE
        )

        user.set_password(data['password'])

        # Установка временных ограничений
        if data.get('access_start_time'):
            user.access_start_time =
datetime.strptime(data['access_start_time'], '%H:%M').time()
        if data.get('access_end_time'):
            user.access_end_time = datetime.strptime(data['access_end_time'],
'%H:%M').time()

        db.session.add(user)
        db.session.commit()

        # Логирование создания пользователя
        audit_log = AuditLog(
            event_type=AuditEventType.USER_CREATED,
            title='Создан новый пользователь',
            description=f'Создан пользователь {user.username}',
            user_id=session.get('user_id'),
            success=True,
            severity=AuditSeverity.INFO
        )
        db.session.add(audit_log)
        db.session.commit()

        return jsonify({
            'success': True,
            'message': 'Пользователь создан успешно',
```

```
            'user': user.to_dict()
        }), 201

    except Exception as e:
        db.session.rollback()
        return jsonify({'success': False, 'message': f'Ошибка создания
пользователя: {str(e)}'}), 500
```

3. Управление учетными данными (credentials.py)

Модуль управления учетными данными обеспечивает безопасное хранение и доступ к привилегированным учетным данным.

GET /api/credentials

Получает список доступных учетных данных.

Ответ:

```
{
    "success": true,
    "credentials": [
        {
            "id": 1,
            "name": "Production Database",
            "description": "Main production database credentials",
            "credential_type": "database",
            "target_system": "PostgreSQL",
            "target_host": "prod-db.company.com",
            "target_username": "db_admin",
            "status": "active",
            "rotation_enabled": true,
            "next_rotation": "2025-04-01T00:00:00"
        }
    ]
}
```

POST /api/credentials/{id}/access

Запрашивает доступ к учетным данным.

Параметры запроса:

```
{
    "justification": "Emergency database maintenance",
    "duration_minutes": 60
}
```

Логика предоставления доступа:

```python
@credentials_bp.route('/<int:credential_id>/access', methods=['POST'])
@require_auth
def access_credential(credential_id):
    try:
        data = request.get_json()
        user_id = session.get('user_id')

        # Поиск учетных данных
        credential = PrivilegedCredential.query.get_or_404(credential_id)

        # Проверка статуса
        if credential.status != CredentialStatus.ACTIVE:
            return jsonify({'success': False, 'message': 'Учетные данные
недоступны'}), 403

        # Проверка политик доступа
        access_allowed, policy_message = evaluate_access_policies(user_id,
credential)
        if not access_allowed:
            return jsonify({'success': False, 'message': policy_message}), 403

        # Создание записи о доступе
        access_record = CredentialAccess(
            credential_id=credential_id,
            user_id=user_id,
            justification=data.get('justification'),
            duration_minutes=data.get('duration_minutes', 60),
            source_ip=request.remote_addr,
            user_agent=request.headers.get('User-Agent')
        )

        db.session.add(access_record)

        # Логирование доступа
        audit_log = AuditLog(
            event_type=AuditEventType.CREDENTIAL_ACCESS,
            title='Доступ к учетным данным',
            description=f'Пользователь получил доступ к {credential.name}',
            user_id=user_id,
            target_system=credential.target_system,
            target_resource=credential.name,
            success=True,
            severity=AuditSeverity.MEDIUM
        )
        db.session.add(audit_log)
        db.session.commit()

        # Расшифровка учетных данных (только для авторизованного доступа)
        decrypted_password =
credential.decrypt_credential(get_encryption_key())

        return jsonify({
            'success': True,
            'message': 'Доступ предоставлен',
            'access_id': access_record.id,
            'credentials': {
                'username': credential.target_username,
                'password': decrypted_password,
                'host': credential.target_host,
                'port': credential.target_port
            },
```

```
            'expires_at': access_record.expires_at.isoformat()
        })

    except Exception as e:
        db.session.rollback()
        return jsonify({'success': False, 'message': f'Ошибка доступа:
{str(e)}'}), 500
```

4. Управление сессиями (sessions.py)

Модуль управления сессиями отслеживает все привилегированные подключения пользователей.

GET /api/sessions

Получает список активных сессий.

Ответ:

```
{
    "success": true,
    "sessions": [
        {
            "id": 1,
            "session_token": "sess_abc123",
            "user_id": 1,
            "username": "admin",
            "target_system": "Production Server",
            "target_host": "prod-web-01.company.com",
            "session_type": "ssh",
            "status": "active",
            "start_time": "2025-01-01T10:00:00",
            "duration_minutes": 45,
            "source_ip": "192.168.1.100",
            "commands_logged": 23
        }
    ]
}
```

POST /api/sessions

Создает новую привилегированную сессию.

Параметры запроса:

```
{
    "target_system": "Production Server",
    "target_host": "prod-web-01.company.com",
    "target_port": 22,
    "session_type": "ssh",
    "justification": "Server maintenance",
    "max_duration_minutes": 120
}
```

POST /api/sessions/{id}/terminate

Принудительно завершает сессию.

Логика завершения сессии:

```
@sessions_bp.route('/<int:session_id>/terminate', methods=['POST'])
@require_auth
@require_role(['admin', 'operator'])
def terminate_session(session_id):
    try:
        data = request.get_json()
        reason = data.get('reason', 'Manual termination')

        session_obj = PrivilegedSession.query.get_or_404(session_id)

        if session_obj.status != SessionStatus.ACTIVE:
            return jsonify({'success': False, 'message': 'Сессия уже
завершена'}), 400

        # Завершение сессии
        session_obj.terminate(reason)

        # Логирование
        audit_log = AuditLog(
            event_type=AuditEventType.SESSION_END,
            title='Принудительное завершение сессии',
            description=f'Сессия {session_obj.session_token} завершена:
{reason}',
            user_id=session.get('user_id'),
            session_id=session_id,
            success=True,
            severity=AuditSeverity.MEDIUM
        )
        db.session.add(audit_log)
        db.session.commit()

        return jsonify({
            'success': True,
            'message': 'Сессия завершена успешно'
        })

    except Exception as e:
        db.session.rollback()
        return jsonify({'success': False, 'message': f'Ошибка завершения
сессии: {str(e)}'}), 500
```

5. Аудит и отчетность (audit.py)

Модуль аудита предоставляет доступ к журналам событий и генерацию отчетов.

GET /api/audit/logs

Получает журнал аудита с фильтрацией.

Параметры запроса: - `start_date` - начальная дата (ISO формат) - `end_date` - конечная дата - `event_type` - тип события - `severity` - уровень серьезности - `user_id` - ID пользователя - `page` - номер страницы - `per_page` - записей на странице

Ответ:

```
{
    "success": true,
    "logs": [
        {
            "id": 1,
            "event_type": "login",
            "severity": "info",
            "timestamp": "2025-01-01T10:00:00",
            "title": "Успешный вход в систему",
            "description": "Пользователь admin успешно аутентифицирован",
            "user_id": 1,
            "username": "admin",
            "source_ip": "192.168.1.100",
            "success": true
        }
    ],
    "statistics": {
        "total_events": 1000,
        "security_events": 50,
        "failed_logins": 5,
        "policy_violations": 2
    }
}
```

GET /api/audit/statistics

Получает статистику аудита за период.

Ответ:

```
{
    "success": true,
    "statistics": {
        "total_events": 1500,
        "events_by_type": {
            "login": 200,
            "logout": 180,
            "credential_access": 300,
            "session_start": 150,
            "session_end": 145
        },
        "events_by_severity": {
            "info": 1200,
            "low": 150,
            "medium": 100,
            "high": 40,
            "critical": 10
        },
        "top_users": [
            {"username": "admin", "events": 500},
            {"username": "operator", "events": 300}
        ],
        "security_alerts": 25,
        "policy_violations": 8
    }
}
```

6. Политики доступа (policies.py)

Модуль политик управляет правилами доступа к ресурсам системы.

GET /api/policies

Получает список политик доступа.

POST /api/policies/evaluate

Оценивает применимость политик для конкретного запроса доступа.

Параметры запроса:

```
{
    "user_id": 1,
    "target_system": "Production Database",
    "source_ip": "192.168.1.100",
    "requested_time": "2025-01-01T14:00:00"
}
```

Логика оценки политик:

```python
@policies_bp.route('/evaluate', methods=['POST'])
@require_auth
def evaluate_policies():
    try:
        data = request.get_json()
        user_id = data.get('user_id')
        target_system = data.get('target_system')
        source_ip = data.get('source_ip', request.remote_addr)

        # Получение пользователя
        user = PrivilegedUser.query.get(user_id)
        if not user:
            return jsonify({'success': False, 'message': 'Пользователь не
найден'}), 404

        # Получение активных политик, отсортированных по приоритету
        policies =
AccessPolicy.query.filter_by(status=PolicyStatus.ACTIVE).order_by(AccessPolicy.pr

        evaluation_results = []
        final_decision = 'allow'  # По умолчанию разрешаем

        for policy in policies:
            result = evaluate_single_policy(policy, user, target_system,
source_ip)
            evaluation_results.append(result)

            # Применяем первую сработавшую политику
            if result['applies']:
                if result['action'] == 'deny':
                    final_decision = 'deny'
                    break
                elif result['action'] == 'require_approval':
                    final_decision = 'require_approval'
                    break

        return jsonify({
            'success': True,
            'decision': final_decision,
            'policies_evaluated': len(policies),
            'applicable_policies': [r for r in evaluation_results if
r['applies']],
            'evaluation_details': evaluation_results
        })

    except Exception as e:
        return jsonify({'success': False, 'message': f'Ошибка оценки политик:
{str(e)}'}), 500

def evaluate_single_policy(policy, user, target_system, source_ip):
    """Оценивает применимость одной политики"""
    applies = True
    reasons = []

    # Проверка пользователей/ролей
    if policy.target_users:
        target_users = json.loads(policy.target_users)
        if user.id not in target_users and user.username not in target_users:
            applies = False
            reasons.append('Пользователь не в списке целевых пользователей')
```

```
    if policy.target_roles:
        target_roles = json.loads(policy.target_roles)
        if user.role.value not in target_roles:
            applies = False
            reasons.append('Роль пользователя не в списке целевых ролей')

    # Проверка систем
    if policy.target_systems:
        target_systems = json.loads(policy.target_systems)
        if target_system not in target_systems:
            applies = False
            reasons.append('Система не в списке целевых систем')

    # Проверка IP-адресов
    if policy.source_ip_ranges:
        ip_ranges = json.loads(policy.source_ip_ranges)
        if not is_ip_in_ranges(source_ip, ip_ranges):
            applies = False
            reasons.append('IP-адрес не в разрешенных диапазонах')

    # Проверка времени
    if policy.time_start and policy.time_end:
        current_time = datetime.now().time()
        if not (policy.time_start <= current_time <= policy.time_end):
            applies = False
            reasons.append('Текущее время не в разрешенном диапазоне')

    # Проверка дней недели
    if policy.days_of_week:
        allowed_days = [int(d) for d in policy.days_of_week.split(',')]
        current_day = datetime.now().weekday() + 1  # Понедельник = 1
        if current_day not in allowed_days:
            applies = False
            reasons.append('Текущий день недели не разрешен')

    return {
        'policy_id': policy.id,
        'policy_name': policy.name,
        'applies': applies,
        'action': policy.action.value if applies else None,
        'reasons': reasons
    }
```

7. Дашборд (dashboard.py)

Модуль дашборда предоставляет сводную информацию о состоянии системы.

GET /api/dashboard/overview

Получает общий обзор системы.

Ответ:

```
{
    "success": true,
    "overview": {
        "active_sessions": 5,
        "total_users": 25,
        "total_credentials": 50,
        "security_alerts": 3,
        "recent_activities": [
            {
                "timestamp": "2025-01-01T14:30:00",
                "event": "Новый вход в систему",
                "user": "operator",
                "severity": "info"
            }
        ]
    }
}
```


GET /api/dashboard/security-alerts

Получает текущие алерты безопасности.

GET /api/dashboard/system-health

Получает информацию о состоянии системы.

Ответ:


```
{
    "success": true,
    "health": {
        "status": "healthy",
        "uptime": "5 days, 12 hours",
        "database_status": "connected",
        "active_connections": 15,
        "memory_usage": "45%",
        "disk_usage": "60%",
        "last_backup": "2025-01-01T02:00:00"
    }
}
```


Веб-интерфейс

Архитектура фронтенда

Веб-интерфейс PAM-системы реализован как Single Page Application (SPA) с использованием:

- **HTML5** - семантическая разметка
- **CSS3** - стилизация с использованием CSS Grid и Flexbox
- **JavaScript (ES6+)** - логика интерфейса
- **Bootstrap 5** - UI компоненты и адаптивность
- **Font Awesome** - иконки

Основные компоненты интерфейса

1. Форма аутентификации

```
<div class="login-container">
    <div class="login-card">
        <div class="login-header">
            <i class="fas fa-shield-alt"></i>
            <h2>PAM System</h2>
            <p>Управление привилегированным доступом</p>
        </div>
        <form id="loginForm">
            <div class="form-group">
                <i class="fas fa-user"></i>
                <input type="text" id="username" placeholder="Имя пользователя"
required>
            </div>
            <div class="form-group">
                <i class="fas fa-lock"></i>
                <input type="password" id="password" placeholder="Пароль"
required>
            </div>
            <button type="submit" class="btn-login">
                <i class="fas fa-sign-in-alt"></i> Войти
            </button>
        </form>
    </div>
</div>
```

2. Главная навигация

```
<nav class="main-nav">
    <div class="nav-brand">
        <i class="fas fa-shield-alt"></i>
        <span>PAM System</span>
    </div>
    <ul class="nav-menu">
        <li><a href="#dashboard" class="nav-link active">
            <i class="fas fa-tachometer-alt"></i> Дашборд
        </a></li>
        <li><a href="#users" class="nav-link">
            <i class="fas fa-users"></i> Пользователи
        </a></li>
        <li><a href="#credentials" class="nav-link">
            <i class="fas fa-key"></i> Учетные данные
        </a></li>
        <li><a href="#sessions" class="nav-link">
            <i class="fas fa-desktop"></i> Сессии
        </a></li>
        <li><a href="#policies" class="nav-link">
            <i class="fas fa-shield-alt"></i> Политики
        </a></li>
        <li><a href="#audit" class="nav-link">
            <i class="fas fa-clipboard-list"></i> Аудит
        </a></li>
    </ul>
    <div class="user-menu">
        <span id="currentUser"></span>
        <button onclick="logout()">
            <i class="fas fa-sign-out-alt"></i> Выход
        </button>
    </div>
</nav>
```

3. Дашборд

```
<div id="dashboard" class="content-section">
    <h1>Дашборд</h1>

    <div class="stats-grid">
        <div class="stat-card">
            <div class="stat-icon">
                <i class="fas fa-users"></i>
            </div>
            <div class="stat-info">
                <h3 id="totalUsers">0</h3>
                <p>Всего пользователей</p>
            </div>
        </div>

        <div class="stat-card">
            <div class="stat-icon">
                <i class="fas fa-desktop"></i>
            </div>
            <div class="stat-info">
                <h3 id="activeSessions">0</h3>
                <p>Активные сессии</p>
            </div>
        </div>

        <div class="stat-card alert">
            <div class="stat-icon">
                <i class="fas fa-exclamation-triangle"></i>
            </div>
            <div class="stat-info">
                <h3 id="securityAlerts">0</h3>
                <p>Алерты безопасности</p>
            </div>
        </div>
    </div>

    <div class="dashboard-grid">
        <div class="dashboard-card">
            <h3>Последние события</h3>
            <div id="recentEvents" class="events-list">
                <!-- События загружаются динамически -->
            </div>
        </div>

        <div class="dashboard-card">
            <h3>Состояние системы</h3>
            <div id="systemHealth" class="health-indicators">
                <!-- Индикаторы загружаются динамически -->
            </div>
        </div>
    </div>
</div>
```

JavaScript функциональность

Управление аутентификацией

```
class AuthManager {
    constructor() {
        this.currentUser = null;
        this.sessionToken = null;
    }

    async login(username, password) {
        try {
            const response = await fetch('/api/auth/login', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({ username, password })
            });

            const data = await response.json();

            if (data.success) {
                this.currentUser = data.user;
                this.sessionToken = data.session_token;
                localStorage.setItem('sessionToken', this.sessionToken);
                this.showMainInterface();
                return true;
            } else {
                this.showError(data.message);
                return false;
            }
        } catch (error) {
            this.showError('Ошибка соединения с сервером');
            return false;
        }
    }

    async logout() {
        try {
            await fetch('/api/auth/logout', {
                method: 'POST',
                headers: {
                    'Authorization': `Bearer ${this.sessionToken}`
                }
            });
        } catch (error) {
            console.error('Ошибка при выходе:', error);
        }

        this.currentUser = null;
        this.sessionToken = null;
        localStorage.removeItem('sessionToken');
        this.showLoginForm();
    }

    async checkSession() {
        const token = localStorage.getItem('sessionToken');
        if (!token) return false;
```

```
        try {
            const response = await fetch('/api/auth/check-session', {
                headers: {
                    'Authorization': `Bearer ${token}`
                }
            });

            const data = await response.json();

            if (data.authenticated) {
                this.currentUser = data.user;
                this.sessionToken = token;
                return true;
            }
        } catch (error) {
            console.error('Ошибка проверки сессии:', error);
        }

        return false;
    }
}
```

Управление данными

```
class DataManager {
    constructor(authManager) {
        this.auth = authManager;
    }

    async apiCall(endpoint, options = {}) {
        const defaultOptions = {
            headers: {
                'Content-Type': 'application/json',
                'Authorization': `Bearer ${this.auth.sessionToken}`
            }
        };

        const mergedOptions = {
            ...defaultOptions,
            ...options,
            headers: {
                ...defaultOptions.headers,
                ...options.headers
            }
        };

        try {
            const response = await fetch(endpoint, mergedOptions);

            if (response.status === 401) {
                this.auth.logout();
                return null;
            }

            return await response.json();
        } catch (error) {
            console.error('API Error:', error);
            throw error;
        }
    }

    async getUsers(filters = {}) {
        const params = new URLSearchParams(filters);
        return await this.apiCall(`/api/users?${params}`);
    }

    async getCredentials() {
        return await this.apiCall('/api/credentials');
    }

    async getSessions() {
        return await this.apiCall('/api/sessions');
    }

    async getAuditLogs(filters = {}) {
        const params = new URLSearchParams(filters);
        return await this.apiCall(`/api/audit/logs?${params}`);
    }

    async getDashboardData() {
        return await this.apiCall('/api/dashboard/overview');
```

```
    }
}
```

Управление интерфейсом

```
class UIManager {
    constructor(authManager, dataManager) {
        this.auth = authManager;
        this.data = dataManager;
        this.currentSection = 'dashboard';
    }

    init() {
        this.setupEventListeners();
        this.setupNavigation();
    }

    setupEventListeners() {
        // Обработка формы входа
        document.getElementById('loginForm').addEventListener('submit', async
(e) => {
            e.preventDefault();
            const username = document.getElementById('username').value;
            const password = document.getElementById('password').value;
            await this.auth.login(username, password);
        });

        // Обработка навигации
        document.querySelectorAll('.nav-link').forEach(link => {
            link.addEventListener('click', (e) => {
                e.preventDefault();
                const section = link.getAttribute('href').substring(1);
                this.showSection(section);
            });
        });
    }

    async showSection(sectionName) {
        // Скрыть все секции
        document.querySelectorAll('.content-section').forEach(section => {
            section.style.display = 'none';
        });

        // Обновить активную навигацию
        document.querySelectorAll('.nav-link').forEach(link => {
            link.classList.remove('active');
        });

document.querySelector(`[href="#${sectionName}"]`).classList.add('active');

        // Показать выбранную секцию
        const section = document.getElementById(sectionName);
        section.style.display = 'block';

        // Загрузить данные для секции
        await this.loadSectionData(sectionName);

        this.currentSection = sectionName;
    }

    async loadSectionData(sectionName) {
        switch (sectionName) {
            case 'dashboard':
                await this.loadDashboard();
```

```
                break;
            case 'users':
                await this.loadUsers();
                break;
            case 'credentials':
                await this.loadCredentials();
                break;
            case 'sessions':
                await this.loadSessions();
                break;
            case 'audit':
                await this.loadAuditLogs();
                break;
        }
    }

    async loadDashboard() {
        try {
            const data = await this.data.getDashboardData();
            if (data && data.success) {
                this.updateDashboardStats(data.overview);
            }
        } catch (error) {
            console.error('Ошибка загрузки дашборда:', error);
        }
    }

    updateDashboardStats(overview) {
        document.getElementById('totalUsers').textContent =
overview.total_users || 0;
        document.getElementById('activeSessions').textContent =
overview.active_sessions || 0;
        document.getElementById('securityAlerts').textContent =
overview.security_alerts || 0;

        // Обновление списка последних событий
        const eventsContainer = document.getElementById('recentEvents');
        eventsContainer.innerHTML = '';

        if (overview.recent_activities) {
            overview.recent_activities.forEach(activity => {
                const eventElement = document.createElement('div');
                eventElement.className = 'event-item';
                eventElement.innerHTML = `
                    <div class="event-time">${new
Date(activity.timestamp).toLocaleString()}</div>
                    <div class="event-description">${activity.event}</div>
                    <div class="event-user">Пользователь: ${activity.user}
</div>
                `;
                eventsContainer.appendChild(eventElement);
            });
        }
    }
}
```

Безопасность системы

Шифрование данных

Шифрование учетных данных

Все привилегированные учетные данные шифруются с использованием симметричного шифрования Fernet (AES 128 в режиме CBC):

```
 from cryptography.fernet import Fernet
import os

class EncryptionManager:
    def __init__(self):
        self.key = self._get_or_create_key()
        self.fernet = Fernet(self.key)

    def _get_or_create_key(self):
        """Получает или создает ключ шифрования"""
        key_file = 'encryption.key'

        if os.path.exists(key_file):
            with open(key_file, 'rb') as f:
                return f.read()
        else:
            key = Fernet.generate_key()
            with open(key_file, 'wb') as f:
                f.write(key)
            return key

    def encrypt(self, data):
        """Шифрует данные"""
        if isinstance(data, str):
            data = data.encode()
        return self.fernet.encrypt(data)

    def decrypt(self, encrypted_data):
        """Расшифровывает данные"""
        if isinstance(encrypted_data, str):
            encrypted_data = encrypted_data.encode()
        decrypted = self.fernet.decrypt(encrypted_data)
        return decrypted.decode()
```

Хеширование паролей

Пароли пользователей хешируются с использованием bcrypt с солью:

```python
from werkzeug.security import generate_password_hash, check_password_hash

def hash_password(password):
    """Создает хеш пароля с солью"""
    return generate_password_hash(password, method='pbkdf2:sha256',
salt_length=8)

def verify_password(password, password_hash):
    """Проверяет пароль против хеша"""
    return check_password_hash(password_hash, password)
```

Аутентификация и авторизация

Сессионная аутентификация

Система использует сессионную аутентификацию с токенами:

```python
import secrets
from datetime import datetime, timedelta

class SessionManager:
    def __init__(self):
        self.active_sessions = {}
        self.session_timeout = timedelta(hours=8)

    def create_session(self, user_id):
        """Создает новую сессию"""
        session_token = secrets.token_urlsafe(32)
        session_data = {
            'user_id': user_id,
            'created_at': datetime.utcnow(),
            'last_activity': datetime.utcnow(),
            'expires_at': datetime.utcnow() + self.session_timeout
        }

        self.active_sessions[session_token] = session_data
        return session_token

    def validate_session(self, session_token):
        """Проверяет валидность сессии"""
        if session_token not in self.active_sessions:
            return None

        session_data = self.active_sessions[session_token]

        # Проверка истечения сессии
        if datetime.utcnow() > session_data['expires_at']:
            del self.active_sessions[session_token]
            return None

        # Обновление времени последней активности
        session_data['last_activity'] = datetime.utcnow()

        return session_data

    def terminate_session(self, session_token):
        """Завершает сессию"""
        if session_token in self.active_sessions:
            del self.active_sessions[session_token]
```

Ролевая авторизация

Система использует декораторы для контроля доступа:

```python
from functools import wraps
from flask import session, jsonify

def require_auth(f):
    """Декоратор для требования аутентификации"""
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user_id' not in session:
            return jsonify({'success': False, 'message': 'Требуется
аутентификация'}), 401
        return f(*args, **kwargs)
    return decorated_function

def require_role(allowed_roles):
    """Декоратор для требования определенной роли"""
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            user_id = session.get('user_id')
            if not user_id:
                return jsonify({'success': False, 'message': 'Требуется
аутентификация'}), 401

            user = PrivilegedUser.query.get(user_id)
            if not user or user.role.value not in allowed_roles:
                return jsonify({'success': False, 'message': 'Недостаточно прав
доступа'}), 403

            return f(*args, **kwargs)
        return decorated_function
    return decorator

# Использование декораторов
@users_bp.route('', methods=['POST'])
@require_auth
@require_role(['admin'])
def create_user():
    # Только администраторы могут создавать пользователей
    pass
```

Защита от атак

CSRF защита

```python
from flask_wtf.csrf import CSRFProtect

csrf = CSRFProtect()
csrf.init_app(app)

# Все формы должны включать CSRF токен
# <input type="hidden" name="csrf_token" value="{{ csrf_token() }}"/>
```

Rate Limiting

```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["200 per day", "50 per hour"]
)

@auth_bp.route('/login', methods=['POST'])
@limiter.limit("5 per minute")
def login():
    # Ограничение попыток входа
    pass
```

SQL Injection защита

Использование SQLAlchemy ORM предотвращает SQL инъекции:

```python
# Безопасный запрос через ORM
user = PrivilegedUser.query.filter_by(username=username).first()

# Вместо небезопасного:
# cursor.execute(f"SELECT * FROM users WHERE username = '{username}'")
```

XSS защита

```python
from markupsafe import escape

def safe_render(template, **kwargs):
    """Безопасный рендеринг с экранированием"""
    for key, value in kwargs.items():
        if isinstance(value, str):
            kwargs[key] = escape(value)
    return render_template(template, **kwargs)
```

Аудит безопасности

Логирование событий безопасности

```python
 def log_security_event(event_type, description, user_id=None,
severity='medium', **kwargs):
    """Логирует события безопасности"""
    audit_log = AuditLog(
        event_type=event_type,
        title=f'Событие безопасности: {event_type.value}',
        description=description,
        user_id=user_id,
        severity=AuditSeverity(severity),
        source_ip=kwargs.get('source_ip'),
        user_agent=kwargs.get('user_agent'),
        success=kwargs.get('success', True)
    )

    # Добавление метаданных
    metadata = {
        'timestamp': datetime.utcnow().isoformat(),
        'additional_info': kwargs.get('additional_info', {})
    }
    audit_log.set_metadata(metadata)

    db.session.add(audit_log)
    db.session.commit()

    # Отправка алертов для критических событий
    if severity in ['high', 'critical']:
        send_security_alert(audit_log)

def send_security_alert(audit_log):
    """Отправляет алерт безопасности"""
    alert = SecurityAlert(
        title=audit_log.title,
        description=audit_log.description,
        severity=audit_log.severity,
        event_id=audit_log.id,
        triggered_at=datetime.utcnow()
    )

    # Отправка уведомлений администраторам
    admins = PrivilegedUser.query.filter_by(role=UserRole.ADMIN).all()
    for admin in admins:
        send_notification(admin.email, alert)
```

Обнаружение аномалий

```
class AnomalyDetector:
    def __init__(self):
        self.baseline_patterns = {}

    def analyze_login_pattern(self, user_id, source_ip, timestamp):
        """Анализирует паттерны входа пользователя"""
        user_key = f"user_{user_id}"

        if user_key not in self.baseline_patterns:
            self.baseline_patterns[user_key] = {
                'usual_ips': set(),
                'usual_times': [],
                'login_frequency': {}
            }

        pattern = self.baseline_patterns[user_key]

        # Проверка необычного IP
        if source_ip not in pattern['usual_ips']:
            if len(pattern['usual_ips']) > 0:  # Не первый вход
                self.trigger_anomaly_alert(
                    user_id,
                    'unusual_ip',
                    f'Вход с нового IP-адреса: {source_ip}'
                )

        pattern['usual_ips'].add(source_ip)

        # Проверка необычного времени
        hour = timestamp.hour
        if hour not in pattern['usual_times']:
            if len(pattern['usual_times']) > 5:  # Достаточно данных для
анализа
                avg_hour = sum(pattern['usual_times']) /
len(pattern['usual_times'])
                if abs(hour - avg_hour) > 4:  # Отклонение более 4 часов
                    self.trigger_anomaly_alert(
                        user_id,
                        'unusual_time',
                        f'Вход в необычное время: {hour}:00'
                    )

        pattern['usual_times'].append(hour)

        # Ограничиваем размер истории
        if len(pattern['usual_times']) > 100:
            pattern['usual_times'] = pattern['usual_times'][-50:]

    def trigger_anomaly_alert(self, user_id, anomaly_type, description):
        """Создает алерт об аномалии"""
        log_security_event(
            AuditEventType.ANOMALY_DETECTED,
            description,
            user_id=user_id,
            severity='medium',
            additional_info={'anomaly_type': anomaly_type}
        )
```

Развертывание и эксплуатация

Требования к системе

Минимальные требования

- **ОС**: Ubuntu 20.04+ / CentOS 8+ / Windows Server 2019+
- **Python**: 3.8+
- **RAM**: 2 GB
- **Диск**: 10 GB свободного места
- **Сеть**: HTTP/HTTPS доступ

Рекомендуемые требования

- **ОС**: Ubuntu 22.04 LTS
- **Python**: 3.11+
- **RAM**: 8 GB
- **Диск**: 50 GB SSD
- **База данных**: PostgreSQL 14+
- **Веб-сервер**: Nginx + Gunicorn
- **SSL**: Сертификат от доверенного CA

Установка и настройка

Установка зависимостей

```
# Обновление системы
sudo apt update && sudo apt upgrade -y

# Установка Python и pip
sudo apt install python3.11 python3.11-venv python3-pip -y

# Установка PostgreSQL
sudo apt install postgresql postgresql-contrib -y

# Создание пользователя и базы данных
sudo -u postgres createuser pamuser
sudo -u postgres createdb pamdb -O pamuser
sudo -u postgres psql -c "ALTER USER pamuser PASSWORD 'secure_password';"

# Установка Nginx
sudo apt install nginx -y

# Установка Redis (для кеширования)
sudo apt install redis-server -y
```

Настройка приложения

```
# Клонирование репозитория
git clone https://github.com/company/pam-system.git
cd pam-system

# Создание виртуального окружения
python3.11 -m venv venv
source venv/bin/activate

# Установка зависимостей
pip install -r requirements.txt

# Создание файла конфигурации
cp config.example.py config.py
```

Конфигурационный файл

```
# config.py
import os

class Config:
    # Основные настройки
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'your-secret-key-here'

    # База данных
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'postgresql://pamuser:secure_password@localhost/pamdb'
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    # Безопасность
    SESSION_COOKIE_SECURE = True
    SESSION_COOKIE_HTTPONLY = True
    SESSION_COOKIE_SAMESITE = 'Lax'
    PERMANENT_SESSION_LIFETIME = 28800  # 8 часов

    # Шифрование
    ENCRYPTION_KEY_FILE = os.environ.get('ENCRYPTION_KEY_FILE') or
'encryption.key'

    # Логирование
    LOG_LEVEL = os.environ.get('LOG_LEVEL') or 'INFO'
    LOG_FILE = os.environ.get('LOG_FILE') or '/var/log/pam-system/app.log'

    # Email уведомления
    MAIL_SERVER = os.environ.get('MAIL_SERVER')
    MAIL_PORT = int(os.environ.get('MAIL_PORT') or 587)
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS', 'true').lower() in ['true',
'on', '1']
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')

    # Rate limiting
    RATELIMIT_STORAGE_URL = os.environ.get('REDIS_URL') or
'redis://localhost:6379'

class ProductionConfig(Config):
    DEBUG = False
    TESTING = False

class DevelopmentConfig(Config):
    DEBUG = True
    TESTING = False

class TestingConfig(Config):
    DEBUG = True
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:///:memory:'

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}
```

Systemd сервис

```
# /etc/systemd/system/pam-system.service
[Unit]
Description=PAM System
After=network.target postgresql.service

[Service]
Type=exec
User=pamuser
Group=pamuser
WorkingDirectory=/opt/pam-system
Environment=FLASK_ENV=production
Environment=DATABASE_URL=postgresql://pamuser:secure_password@localhost/pamdb
ExecStart=/opt/pam-system/venv/bin/gunicorn --bind 127.0.0.1:5000 --workers 4
main:app
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Nginx конфигурация

```
# /etc/nginx/sites-available/pam-system
server {
    listen 80;
    server_name pam.company.com;
    return 301 https://$`server_name`$request_uri;
}

server {
    listen 443 ssl http2;
    server_name pam.company.com;

    ssl_certificate /etc/ssl/certs/pam.company.com.crt;
    ssl_certificate_key /etc/ssl/private/pam.company.com.key;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-
RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384;
    ssl_prefer_server_ciphers off;

    location / {
        proxy_pass http://127.0.0.1:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    location /static {
        alias /opt/pam-system/src/static;
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
```

Мониторинг и обслуживание

Логирование

```
 import logging
from logging.handlers import RotatingFileHandler

def setup_logging(app):
    if not app.debug:
        if not os.path.exists('logs'):
            os.mkdir('logs')

        file_handler = RotatingFileHandler(
            'logs/pam-system.log',
            maxBytes=10240000,  # 10MB
            backupCount=10
        )

        file_handler.setFormatter(logging.Formatter(
            '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%
(lineno)d]'
        ))

        file_handler.setLevel(logging.INFO)
        app.logger.addHandler(file_handler)
        app.logger.setLevel(logging.INFO)
        app.logger.info('PAM System startup')
```

Резервное копирование

```
 #!/bin/bash
# backup.sh - Скрипт резервного копирования

BACKUP_DIR="/opt/backups/pam-system"
DATE=$(date +%Y%m%d_%H%M%S)
DB_BACKUP="$`BACKUP_DIR/db_backup_`$DATE.sql"
FILES_BACKUP="$`BACKUP_DIR/files_backup_`$DATE.tar.gz"

# Создание директории для бэкапов
mkdir -p $BACKUP_DIR

# Резервное копирование базы данных
pg_dump -h localhost -U pamuser pamdb > $DB_BACKUP

# Резервное копирование файлов
tar -czf $FILES_BACKUP \
    /opt/pam-system/config.py \
    /opt/pam-system/encryption.key \
    /opt/pam-system/logs/

# Удаление старых бэкапов (старше 30 дней)
find $BACKUP_DIR -name "*.sql" -mtime +30 -delete
find $BACKUP_DIR -name "*.tar.gz" -mtime +30 -delete

echo "Backup completed: $DATE"
```

Мониторинг производительности

```
 from flask import g
import time
import psutil

@app.before_request
def before_request():
    g.start_time = time.time()

@app.after_request
def after_request(response):
    # Время выполнения запроса
    execution_time = time.time() - g.start_time

    # Логирование медленных запросов
    if execution_time > 1.0:  # > 1 секунды
        app.logger.warning(f'Slow request: {request.endpoint} took
{execution_time:.2f}s')

    # Добавление заголовка с временем выполнения
    response.headers['X-Response-Time'] = f'{execution_time:.3f}s'

    return response

def get_system_metrics():
    """Получает метрики системы"""
    return {
        'cpu_percent': psutil.cpu_percent(),
        'memory_percent': psutil.virtual_memory().percent,
        'disk_percent': psutil.disk_usage('/').percent,
        'active_connections': len(psutil.net_connections()),
        'uptime': time.time() - psutil.boot_time()
    }
```

Заключение

Разработанная PAM-система представляет собой комплексное решение для управления привилегированным доступом, которое обеспечивает высокий уровень безопасности при сохранении удобства использования. Система реализует все основные функции PAM-решения: централизованное управление учетными данными, мониторинг сессий, контроль доступа через политики, полный аудит действий пользователей.

Архитектура системы позволяет легко масштабировать и расширять функциональность, а использование современных технологий и принципов безопасности обеспечивает надежную защиту критически важных ресурсов организации.

Система готова к развертыванию в продакшене и может быть адаптирована под специфические требования различных организаций.