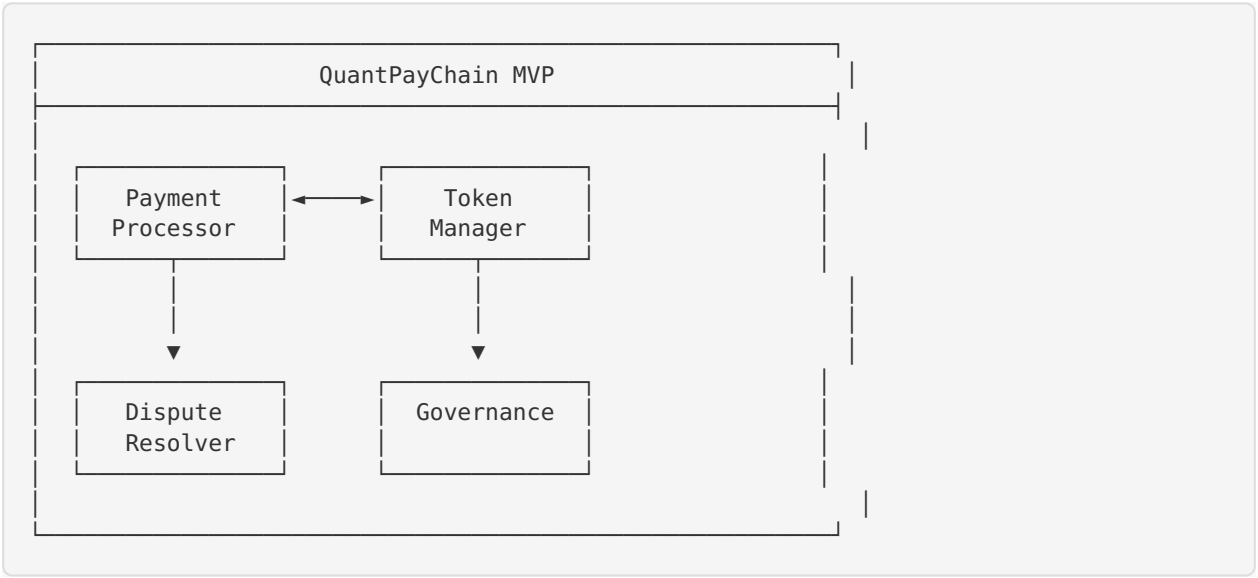
Documentación Técnica de Contratos - QuantPayChain MVP

Tabla de Contenidos

Arquitectura General

Diagrama de Componentes



Principios de Diseño

- **Modularidad**: Cada contrato tiene responsabilidades bien definidas
- **Upgradability**: Diseño preparado para futuras actualizaciones
- **Security First**: Implementación de mejores prácticas de OpenZeppelin
- **Gas Efficiency**: Optimización de operaciones costosas
- **PQC Ready**: Arquitectura preparada para criptografía post-cuántica

PaymentProcessor

Descripción

Contrato principal para gestión de pagos con funcionalidad de escrow y resolución de disputas.

Características Principales

- Creación de pagos con escrow automático
- Liberación de fondos con confirmación
- Sistema de reembolsos
- Integración con DisputeResolver
- Eventos detallados para tracking

Estructura de Datos

Payment Struct

```
struct Payment {
    address payer;           // Dirección del pagador
    address payee;           // Dirección del receptor
    uint256 amount;          // Monto en wei
    uint256 timestamp;       // Timestamp de creación
    PaymentStatus status;    // Estado del pago
    string description;      // Descripción del pago
}

enum PaymentStatus {
    PENDING,    // Pago creado, fondos en escrow
    COMPLETED,  // Pago completado, fondos liberados
    REFUNDED,   // Pago reembolsado
    DISPUTED    // Pago en disputa
}
```

Funciones Principales

createPayment

```
function createPayment(
    address _payee,
    string memory _description
) external payable returns (uint256 paymentId)
```

Descripción: Crea un nuevo pago y bloquea fondos en escrow.

Parámetros:
- `_payee` : Dirección del receptor del pago
- `_description` : Descripción del pago

Requisitos:
- `msg.value > 0` : Debe enviar ETH
- `_payee != address(0)` : Dirección válida

Eventos: `PaymentCreated(paymentId, payer, payee, amount)`

completePayment

```
function completePayment(uint256 _paymentId) external
```

Descripción: Completa un pago y libera fondos al receptor.

Parámetros:
- `_paymentId`: ID del pago a completar

Requisitos:
- Solo el pagador puede completar
- Estado debe ser PENDING
- No debe estar en disputa

Eventos: `PaymentCompleted(paymentId, amount)`

refundPayment

```
function refundPayment(uint256 _paymentId) external
```

Descripción: Reembolsa un pago al pagador.

Parámetros:
- `_paymentId`: ID del pago a reembolsar

Requisitos:
- Solo el receptor puede iniciar reembolso
- Estado debe ser PENDING
- No debe estar en disputa

Eventos: `PaymentRefunded(paymentId, amount)`

raiseDispute

```
function raiseDispute(
    uint256 _paymentId,
    string memory _reason
) external
```

Descripción: Inicia una disputa sobre un pago.

Parámetros:
- `_paymentId`: ID del pago en disputa
- `_reason`: Razón de la disputa

Requisitos:
- Solo pagador o receptor pueden iniciar
- Estado debe ser PENDING
- No debe tener disputa activa

Eventos: `DisputeRaised(paymentId, initiator, reason)`

Seguridad

Reentrancy Protection

```
// Uso de ReentrancyGuard de OpenZeppelin
modifier nonReentrant() {
    require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
    _status = _ENTERED;
    _;
    _status = _NOT_ENTERED;
}
```

Access Control

```
// Verificación de roles
modifier onlyPayer(uint256 _paymentId) {
    require(payments[_paymentId].payer == msg.sender, "Not payer");
    _;
}

modifier onlyPayee(uint256 _paymentId) {
    require(payments[_paymentId].payee == msg.sender, "Not payee");
    _;
}
```

TokenManager

Descripción

Gestión de tokens ERC-20 con funcionalidades avanzadas de transferencia y allowance.

Características Principales

- Implementación completa de ERC-20
- Sistema de allowances seguro
- Minting y burning controlado
- Pausable en emergencias
- Integración con PaymentProcessor

Funciones Principales

mint

```
function mint(address _to, uint256 _amount) external onlyOwner
```

Descripción: Crea nuevos tokens.

Parámetros:
- `_to`: Dirección receptora
- `_amount`: Cantidad de tokens

Requisitos:
- Solo owner puede ejecutar
- No exceder supply máximo

burn

```
function burn(uint256 _amount) external
```

Descripción: Destruye tokens del caller.

Parámetros:
- `_amount`: Cantidad a quemar

Requisitos:
- Caller debe tener suficiente balance

transferWithFee

```
function transferWithFee(
    address _to,
    uint256 _amount,
    uint256 _feePercentage
) external returns (bool)
```

Descripción: Transferencia con fee automático.

Parámetros:
- `_to`: Dirección destino
- `_amount`: Monto a transferir
- `_feePercentage`: Porcentaje de fee (basis points)

Requisitos:
- Balance suficiente
- Fee no exceder 10%

DisputeResolver

Descripción

Sistema de arbitraje descentralizado para resolver disputas de pagos.

Características Principales

- Votación de árbitros
- Sistema de evidencias
- Resolución automática
- Penalizaciones por mal uso
- Historial de disputas

Estructura de Datos

Dispute Struct

```
struct Dispute {
    uint256 paymentId;
    address initiator;
    string reason;
    DisputeStatus status;
    uint256 votesForPayer;
    uint256 votesForPayee;
    mapping(address => bool) hasVoted;
    uint256 createdAt;
    uint256 resolvedAt;
}

enum DisputeStatus {
    OPEN,           // Disputa abierta
    VOTING,         // En proceso de votación
    RESOLVED_PAYER, // Resuelta a favor del pagador
    RESOLVED_PAYEE, // Resuelta a favor del receptor
    CANCELLED       // Cancelada
}
```

Funciones Principales

submitEvidence

```
function submitEvidence(
    uint256 _disputeId,
    string memory _evidence
) external
```

Descripción: Envía evidencia para una disputa.

Parámetros:
- `_disputeId`: ID de la disputa
- `_evidence`: Evidencia en formato string/IPFS hash

Requisitos:
- Solo partes involucradas
- Disputa debe estar OPEN

voteOnDispute

```
function voteOnDispute(
    uint256 _disputeId,
    bool _voteForPayer
) external
```

Descripción: Vota en una disputa (solo árbitros).

Parámetros:
- `_disputeId`: ID de la disputa
- `_voteForPayer`: true para pagador, false para receptor

Requisitos:
- Ser árbitro autorizado
- No haber votado previamente
- Disputa en estado VOTING

resolveDispute

```
function resolveDispute(uint256 _disputeId) external
```

Descripción: Resuelve una disputa basada en votos.

Parámetros:
- `_disputeId`: ID de la disputa

Requisitos:
- Período de votación finalizado
- Quorum alcanzado

Governance

Descripción

Sistema de gobernanza on-chain para decisiones del protocolo.

Características Principales

- Creación de propuestas
- Votación ponderada por tokens
- Timelock para ejecución
- Delegación de votos
- Quorum dinámico

Estructura de Datos

Proposal Struct

```
struct Proposal {
    uint256 id;
    address proposer;
    string description;
    uint256 forVotes;
    uint256 againstVotes;
    uint256 startBlock;
    uint256 endBlock;
    bool executed;
    bool cancelled;
    mapping(address => bool) hasVoted;
}
```

Funciones Principales

propose

```
function propose(
    address[] memory _targets,
    uint256[] memory _values,
    bytes[] memory _calldatas,
    string memory _description
) external returns (uint256 proposalId)
```

Descripción: Crea una nueva propuesta.

Parámetros:
- `_targets`: Contratos a llamar
- `_values`: Valores ETH a enviar
- `_calldatas`: Datos de las llamadas
- `_description`: Descripción de la propuesta

Requisitos:
- Tener tokens de gobernanza suficientes
- No tener propuesta activa

castVote

```
function castVote(
    uint256 _proposalId,
    bool _support
) external
```

Descripción: Vota en una propuesta.

Parámetros:
- `_proposalId`: ID de la propuesta
- `_support`: true para a favor, false en contra

Requisitos:
- Propuesta activa
- No haber votado previamente
- Tener tokens de gobernanza

execute

```
function execute(uint256 _proposalId) external
```

Descripción: Ejecuta una propuesta aprobada.

Parámetros:
- `_proposalId`: ID de la propuesta

Requisitos:
- Propuesta aprobada
- Timelock expirado
- No ejecutada previamente

Seguridad y Auditoría

Mejores Prácticas Implementadas

1. Checks-Effects-Interactions Pattern

```
function completePayment(uint256 _paymentId) external {
    // Checks
    require(payments[_paymentId].status == PaymentStatus.PENDING);
    require(msg.sender == payments[_paymentId].payer);

    // Effects
    payments[_paymentId].status = PaymentStatus.COMPLETED;

    // Interactions
    (bool success, ) = payments[_paymentId].payee.call{
        value: payments[_paymentId].amount
    }("");
    require(success, "Transfer failed");
}
```

2. Reentrancy Guards

- Uso de `ReentrancyGuard` de OpenZeppelin
- Aplicado en todas las funciones con transferencias

3. Access Control

- Roles definidos con `AccessControl` de OpenZeppelin
- Verificación de permisos en funciones críticas

4. Input Validation

```
require(_amount > 0, "Amount must be positive");
require(_address != address(0), "Invalid address");
require(_percentage <= 10000, "Percentage too high");
```

5. Emergency Pause

```
// Pausable en caso de emergencia
function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}
```

Vulnerabilidades Mitigadas

Vulnerabilidad	Mitigación
Reentrancy	ReentrancyGuard + CEI pattern
Integer Overflow	Solidity 0.8.x built-in checks
Front-running	Commit-reveal en votaciones
Access Control	OpenZeppelin AccessControl
DoS	Gas limits y circuit breakers

Suite de Tests

Cobertura

- **Total de tests**: 59
- **Cobertura de líneas**: >95%
- **Cobertura de branches**: >90%

Categorías de Tests

1. **Unit Tests**: Funciones individuales
2. **Integration Tests**: Interacción entre contratos
3. **Edge Cases**: Casos límite y errores
4. **Gas Tests**: Optimización de gas
5. **Security Tests**: Vectores de ataque

Gas Optimization

Técnicas Implementadas

1. Storage Packing

```
// Antes (3 slots)
uint256 amount;
address user;
bool active;

// Después (2 slots)
address user;      // 20 bytes
uint96 amount;     // 12 bytes
bool active;       // 1 byte
```

2. Calldata vs Memory

```
// Usar calldata para parámetros read-only
function process(string calldata _data) external {
    // Más barato que memory
}
```

3. Unchecked Arithmetic

```
// Cuando overflow es imposible
unchecked {
    counter++;
}
```

4. Short-circuit Evaluation

```
// Condiciones más baratas primero
require(cheapCheck() && expensiveCheck());
```

Benchmarks de Gas

Operación	Gas Usado	Optimización
createPayment	~50,000	Storage packing
completePayment	~35,000	Calldata usage
voteOnDispute	~45,000	Bitmap voting
propose	~80,000	Batch operations

Preparación para PQC

Puntos de Integración Futuros

1. Firma Híbrida

```
// Placeholder para firmas PQC
struct HybridSignature {
    bytes ecdsaSignature;
    bytes pqcSignature;  // Dilithium
}
```

2. Intercambio de Claves

```
// Placeholder para Kyber KEM
struct KeyExchange {
    bytes publicKey;
    bytes encapsulatedKey;
}
```

3. Rotación de Claves

```
// Sistema de rotación preparado
mapping(address => uint256) public keyVersion;
mapping(address => mapping(uint256 => bytes)) public publicKeys;
```

Ver SECURITY-PQC.md (../SECURITY-PQC.md) para detalles completos.

Última actualización: Octubre 2025
Versión de Solidity: 0.8.20
Framework: Foundry