Grupo 01: Miguel Antonio Amato Hermo y Jorge Ortega Carretero

Proyecto Final TBC

Hemos realizado una votación cuadrática como proyecto final de la asignatura de Tecnologías Blockchain y Smart Contracts. Hemos implementado este proyecto en un archivo que contiene 3 contratos: **ExecProposal** que hereda de IExecutableProposal y **myERC20** que hereda de de ERC20, **QuadraticVoting** que implementa la en mayor parte de la votación. En está memoria describiremos el funcionamiento de cada contrato con mayor detalle.

ExecProposal:

Este contrato se trata de un contrato externo el cuál implementa la interfaz IExecutableProposal creada en el propio fichero final.sol, teniendo así una función executeProposal que se encarga de emitir un evento con la información de la propuesta y recibiendo el ETH de está.

myERC20:

Este contrato hereda el código de ERC20 (OpenZeppelin), que permite gestionar los tokens del contrato que se usarán durante el sistema de votación.

En este contrato se crean nuevos tokens mediante la función newTokens(account, tokens) la cuál comprueba si es posible crear el número de tokens recibidos dependiendo únicamente del número máximo de tokens que se puedan crear, dejando la comprobación del precio para ser realizada por otro contrato.

Además, existen dos métodos que permiten eliminar tokens, deleteTokens(account, tokens) que elimina el número de tokens recibidos por parámetro de ese participante, siempre que disponga de suficientes tokens y deleteAllTokens(account) que elimina todos los tokens de una cuenta.

Se ha implementado una función más, checkApprovement(from, to, nTokens) que se encarga de comprobar si hay suficientes tokens aprobados mediante la función approve() de ERC20 antes de realizar una votación.

Grupo 01: Miguel Antonio Amato Hermo y Jorge Ortega Carretero

QuadraticVoting:

Se trata de la parte principal del código del proyecto, en este contrato está implementado el funcionamiento de la votación cuadrática.

Este contrato dispone las siguientes estructuras de datos:

- Una instancia de myERC20 privada gestorToken que es la encargada de gestionar los tokens, su creación, eliminación, aprobación…

- Un Struct Participant que dispone de la información de cada participante:
  - Número de tokens que dispone este participante.
  - Si existe o no el participante.
  - Un mapa en el cual se dispone de una clave el id (uint) de la propuesta y un valor (uint) que es el número de votos de este participante en esa propuesta.

- Otro Struct Proposal que dispone de varios campos:
  - El nombre de la propuesta.
  - La descripción de la propuesta.
  - El presupuesto de la propuesta.
  - El propietario de la propuesta.
  - El número de votos de la propuesta.
  - El número de tokens de la propuesta.
  - Si ha sido aceptada la propuesta.
  - Si ha sido cancelada la propuesta.
  - El umbral de aprobación propuesta.
  - El número de participantes de la propuesta.
  - Un array de direcciones de cada participante de la propuesta.
  - Una instancia addr de IExecutableProposal.

- La dirección del creador del contrato de votación cuadrática.

- El precio en Weis que cuesta cada token.

- El número máximo de tokens en el contrato.

- El presupuesto total de la votación

- El número de propuestas.

- El número de participantes.

- Un array de ids de propuestas aprobadas
- Un array de ids de propuestas financing en estado de espera.
- Un array de ids de propuestas signaling en estado de espera.
- Una variable para saber si la votación se encuentra abierta.
- Un mapa de clave (address) la dirección del participante y valor (Participant) la información del participante con esa dirección teniendo así a todos los participantes representados.
- Un mapa de clave (uint) el id de la propuesta y valor (Proposal) la información de la propuesta, teniendo así todas las propuestas representadas. El id de cada propuesta es único ya que comienza en 0 y se aumenta cada vez que se crea una propuesta. Este identificador se reinicia al cerrar la votación y volver a crear nuevas propuestas tras abrir una nueva votación. Al diseñarse de esta forma podemos recorrer el mapa como si se tratase de un array.

```
myERC20 private gestorToken;

struct Participant {
    uint nTokens;
    bool exist;
    mapping(uint => uint) pVotes; // pId -> votos
}

struct Proposal {
    string name;
    string desc;
    uint256 budget; // presupuesto en ether propuesta
    address owner;
    uint votes;
    uint nTokens;
    bool accepted;
    bool cancel;
    uint threshold;
    uint nParts;
    address[] parts; //participantes
    IExecutableProposal addr;
}

address payable owner;

uint private weiPrice;
uint private nMaxTokens;
uint private totalBudget;
uint private nProposals;
uint private nParticipants;

uint[] financingProposalsPend;
uint[] signalingProposals;
uint[] approvedProposals;

bool open;

mapping (address => Participant) participants; // (address participante -> Participant)
mapping (uint => Proposal) proposals; // (proposalId -> Proposal)
```

Los participantes que quieran registrarse mediante la función addParticipant() mandarán una cantidad de ETH mayor a el precio de un token ya que para participar en la votación mínima se requiere tener un token comprado. El precio del token es gestionado por la instancia de myERC20 **gestorToken** que tiene la función **getWeiPrice()**. Para comprobar que mínimo puede comprar un token se ha creado el modifier **enoughMoneyToBuy().** Al añadirse un participante se añade al mapa **participants** con su dirección, el número de tokens, marcando que existe el participante y con un mapa vacío de propuestas con los votos de este participante a cada una de ellas (por defecto 0).

Para registrar una propuesta mediante la función **addProposal**() envían el nombre de la propuesta, su descripción, el presupuesto de esta propuesta y un address único para está.Comprobamos que esté address no se repita para ver que no existe ya previamente esta propuesta para evitar errores tras interactuar con dicha propuesta.

Para abrir la votación se llama a la función **openVoting**() que recibe el presupuesto total de la votación y sólo puede ser ejecutada por el creador de la votación.

Para cerrar la votación se llama a la función **closeVoting**() que se encarga de cerrar la votación, recorre el array de propuestas signaling que se aprueban y ejecutan al finalizar la votación y se les devuelve los tokens usados para votar a una propuesta signaling a los participantes que lo hayan hecho. También se recorre el array de propuestas financing sin aceptar y se devuelven los tokens a los participantes que las hayan votado.En caso de que haya sobado presupuesto total de la votación, éste será transferido al creador de la votación y se eliminarán todas las propuestas creadas para cuando se empiece una nueva votación.

No importa si está abierta o cerrada la votación, sigue pudiendo registrarse o eliminarse cualquier participante así como comprar o vender tokens.

La función **_checkAndExecuteProposal()** se encarga de comprobar si el balance del contrato supera el presupuesto de la propuesta y si el número de votos supera el umbral para está propuesta en este momento. En caso de superar ambos, se elimina la propuesta de el array de **financingProposalsPend** (solo las financing ya que las signaling se aprueban al cerrar la votación) y se añade al array de propuestas aprobadas **approvedProposals**, se actualiza el presupuesto total y se eliminan los tokens de esta propuesta con el **gestor Token**, trás esto se ejecuta la propuesta y es aceptada.
Está función es llamada en la función **stake()** tras realizar una votación y en la función **withdrawFromProposal()** tras retirar votos de una propuesta; ya que son los casos en los que se actualiza el número de votos para superar el umbral.

Se disponen de tres funciones “getter” que nos devuelven los arrays correspondientes:

- **getPendingProposals()**: devuelve el array de financing proposals en espera.
- **getSignalingProposals()**: devuelve el array de signaling proposals en espera.
- **getApprovedProposals()**: devuelve el array de propuestas aprobadas.

Y una función “getter” que devuelve la información de una propuesta:

- **getProposalInfo():** devuelve el nombre, la descripción, el presupuesto, los votos, el número de tokens, si está aceptada, si está cancelada y el umbral de una propuesta.

Y una función “getter” que devuelve el address del contrato myERC20:

- **getERC20():** devuelve el address de myERC20.

Grupo 01: Miguel Antonio Amato Hermo y Jorge Ortega Carretero

Vulnerabilidades:

- **UnderFlows y OverFlows**: Podemos encontrar una vulnerabilidad de overFlow en la función newTokens() del contrato myERC20 ya que si el número de tokens enviados a crear es muy alto, al sumar el totalSupply() podría ocurrir que volviese a ser 0 o un número inferior a maxTokens y permitir la creación de estos tokens pese a superar el número máximo de tokens creados.
  También existe un overFlow al tratarse de los tokens y los votos, ya que si se vota un número de votos muy grande y al tratarse del coste cuadrático en tokens respecto a los votos, podría llegar a votar una gran cantidad de votos y al superar el límite del uint, gastar 0 o muy pocos tokens. Estos casos son solucionados por Solidity automáticamente al tratarse de una versión superior a la 0.8.20.

- **Parity Wallet:** no existe esta vulnerabilidad en este contrato ya que se crea primero el gestorToken y posteriormente el address del owner.

- **Reentrancy:** no existe esta vulnerabilidad en este contrato ya que se han actualizado los datos previamente y posteriormente se han realizado las transferencias de ETH. No hemos usado send para poder limitar el uso del gas.

- **Denial of Service (Dos):** En caso de añadir una gran cantidad de participantes o de propuestas podría hacer que el contrato fuese inoperable durante un tiempo o incluso permanentemente. Para evitarlo hemos tratado de crear las estructuras de datos lo más óptimas posibles y tratando de recorrer el menor número de datos. Para solucionar este problema se podría haber implementado la solución “favor pull over push” de la parte opcional de la práctica.

- No se usan las funciones **DELEGATECALL, CALLCODE, STATICCALL, CALL** que podrían provocar alguna vulnerabilidad.

- Siempre se usa **msg.sender** en vez de **tx.origin.**

- No hemos empleado **Yul** programming para evitar errores y obtener una mayor legibilidad