Auction.sol

```
// SPDX-License-Identifier: BSD-3-Clause-Clear

pragma solidity >=0.8.13 <0.9.0;

import { Permissioned, Permission } from
"@fhenixprotocol/contracts/access/Permissioned.sol";
import { inEuint32, euint32, FHE } from "@fhenixprotocol/contracts/FHE.sol";
import { euint32, ebool, FHE } from "@fhenixprotocol/contracts/FHE.sol";

pragma solidity ^0.8.20;

// SPDX-License-Identifier: MIT
// Fhenix Protocol (last updated v0.1.0) (token/FHERC20/IFHERC20.sol)
// Inspired by OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts)
(token/ERC20/IERC20.sol)

import { Permission, Permissioned } from
"@fhenixprotocol/contracts/access/Permissioned.sol";
import { euint32, inEuint32 } from "@fhenixprotocol/contracts/FHE.sol";

/**
 * @dev Interface of the ERC-20 standard as defined in the ERC.
 */
interface IFHERC20 {
   /**
    * @dev Emitted when `value` tokens are moved from one account (`from`) to
    * another (`to`).
    *
    * Note that `value` may be zero.
    */
```

```
    event TransferEncrypted(address indexed from, address indexed to);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approveEncrypted}. `value` is the new allowance.
     */
    event ApprovalEncrypted(address indexed owner, address indexed spender);

    // /**
    //  * @dev Returns the value of tokens in existence.
    //  */
    // function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the value of tokens owned by `account`, sealed and encrypted for the caller.
     */
    function balanceOfEncrypted(address account, Permission memory auth) external view
returns (bytes memory);

    /**
     * @dev Moves a `value` amount of tokens from the caller's account to `to`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {TransferEncrypted} event.
     */
    function transferEncrypted(address to, inEuint32 calldata value) external returns (euint32);
    function transferEncrypted(address to, euint32 value) external returns (euint32);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
```

```
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowanceEncrypted(address spender, Permission memory permission) external view returns (bytes memory);

    /**
     * @dev Sets a `value` amount of tokens as the allowance of `spender` over the
     * caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {ApprovalEncrypted} event.
     */
    function approveEncrypted(address spender, inEuint32 calldata value) external returns (bool);

    /**
     * @dev Moves a `value` amount of tokens from `from` to `to` using the
     * allowance mechanism. `value` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
```

```
    *
    * Emits a {TransferEncrypted} event.
    */
   function transferFromEncrypted(address from, address to, inEuint32 calldata value) external returns (euint32);

   function transferFromEncrypted(address from, address to, euint32 value) external returns (euint32);
}


/*
 * @title Solidity Bytes Arrays Utils
 * @author Gonçalo Sá <goncalo.sa@consensys.net>
 *
 * @dev Bytes tightly packed arrays utility library for ethereum contracts written in Solidity.
 *      The library lets you concatenate, slice and type cast bytes arrays both in memory and storage.
 */
pragma solidity >=0.8.13 <0.9.0;


library BytesLib {
   function concat(bytes memory _preBytes, bytes memory _postBytes) internal pure returns (bytes memory) {
       bytes memory tempBytes;

       assembly {
           // Get a location of some free memory and store it in tempBytes as
           // Solidity does for memory variables.
           tempBytes := mload(0x40)

           // Store the length of the first bytes array at the beginning of
           // the memory for tempBytes.
           let length := mload(_preBytes)
```

```
mstore(tempBytes, length)

// Maintain a memory counter for the current write location in the
// temp bytes array by adding the 32 bytes for the array length to
// the starting location.
let mc := add(tempBytes, 0x20)
// Stop copying when the memory counter reaches the length of the
// first bytes array.
let end := add(mc, length)

for {
  // Initialize a copy counter to the start of the _preBytes data,
  // 32 bytes into its memory.
  let cc := add(_preBytes, 0x20)
} lt(mc, end) {
  // Increase both counters by 32 bytes each iteration.
  mc := add(mc, 0x20)
  cc := add(cc, 0x20)
} {
  // Write the _preBytes data into the tempBytes memory 32 bytes
  // at a time.
  mstore(mc, mload(cc))
}

// Add the length of _postBytes to the current length of tempBytes
// and store it as the new length in the first 32 bytes of the
// tempBytes memory.
length := mload(_postBytes)
mstore(tempBytes, add(length, mload(tempBytes)))

// Move the memory counter back from a multiple of 0x20 to the
```

```
        // actual end of the _preBytes data.
        mc := end
        // Stop copying when the memory counter reaches the new combined
        // length of the arrays.
        end := add(mc, length)

        for {
          let cc := add(_postBytes, 0x20)
        } lt(mc, end) {
          mc := add(mc, 0x20)
          cc := add(cc, 0x20)
        } {
          mstore(mc, mload(cc))
        }

        // Update the free-memory pointer by padding our last write location
        // to 32 bytes: add 31 bytes to the end of tempBytes to move to the
        // next 32 byte block, then round down to the nearest multiple of
        // 32. If the sum of the length of the two arrays is zero then add
        // one before rounding down to leave a blank 32 bytes (the length block with 0).
        mstore(
          0x40,
          and(
            add(add(end, iszero(add(length, mload(_preBytes)))), 31),
            not(31) // Round down to the nearest 32 bytes.
          )
        )
      }

      return tempBytes;
    }
```

```
function concatStorage(bytes storage _preBytes, bytes memory _postBytes) internal {
  assembly {
    // Read the first 32 bytes of _preBytes storage, which is the length
    // of the array. (We don't need to use the offset into the slot
    // because arrays use the entire slot.)
    let fslot := sload(_preBytes.slot)
    // Arrays of 31 bytes or less have an even value in their slot,
    // while longer arrays have an odd value. The actual length is
    // the slot divided by two for odd values, and the lowest order
    // byte divided by two for even values.
    // If the slot is even, bitwise and the slot with 255 and divide by
    // two to get the length. If the slot is odd, bitwise and the slot
    // with -1 and divide by two.
    let slength := div(and(fslot, sub(mul(0x100, iszero(and(fslot, 1))), 1)), 2)
    let mlength := mload(_postBytes)
    let newlength := add(slength, mlength)
    // slength can contain both the length and contents of the array
    // if length < 32 bytes so let's prepare for that
    // v. http://solidity.readthedocs.io/en/latest/miscellaneous.html#layout-of-state-
variables-in-storage
    switch add(lt(slength, 32), lt(newlength, 32))
    case 2 {
      // Since the new array still fits in the slot, we just need to
      // update the contents of the slot.
      // uint256(bytes_storage) = uint256(bytes_storage) + uint256(bytes_memory) +
new_length
      sstore(
        _preBytes.slot,
        // all the modifications to the slot are inside this
        // next block
        add(
```

```
        // we can just add to the slot contents because the
        // bytes we want to change are the LSBs
        fslot,
        add(
          mul(
            div(
              // load the bytes from memory
              mload(add(_postBytes, 0x20)),
              // zero all bytes to the right
              exp(0x100, sub(32, mlength))
            ),
            // and now shift left the number of bytes to
            // leave space for the length in the slot
            exp(0x100, sub(32, newlength))
          ),
          // increase length by the double of the memory
          // bytes length
          mul(mlength, 2)
        )
      )
    )
  }
  case 1 {
    // The stored value fits in the slot, but the combined value
    // will exceed it.
    // get the keccak hash to get the contents of the array
    mstore(0x0, _preBytes.slot)
    let sc := add(keccak256(0x0, 0x20), div(slength, 32))

    // save new length
    sstore(_preBytes.slot, add(mul(newlength, 2), 1))
```

```
// The contents of the _postBytes array start 32 bytes into
// the structure. Our first read should obtain the `submod`
// bytes that can fit into the unused space in the last word
// of the stored array. To get this, we read 32 bytes starting
// from `submod`, so the data we read overlaps with the array
// contents by `submod` bytes. Masking the lowest-order
// `submod` bytes allows us to add that value directly to the
// stored value.

let submod := sub(32, slength)
let mc := add(_postBytes, submod)
let end := add(_postBytes, mlength)
let mask := sub(exp(0x100, submod), 1)

sstore(
  sc,
  add(
    and(fslot, 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00),
    and(mload(mc), mask)
  )
)

for {
  mc := add(mc, 0x20)
  sc := add(sc, 1)
} lt(mc, end) {
  sc := add(sc, 1)
  mc := add(mc, 0x20)
} {
  sstore(sc, mload(mc))
```

```
    }

    mask := exp(0x100, sub(mc, end))

    sstore(sc, mul(div(mload(mc), mask), mask))
  }
  default {
    // get the keccak hash to get the contents of the array
    mstore(0x0, _preBytes.slot)
    // Start copying to the last used word of the stored array.
    let sc := add(keccak256(0x0, 0x20), div(slength, 32))

    // save new length
    sstore(_preBytes.slot, add(mul(newlength, 2), 1))

    // Copy over the first `submod` bytes of the new data as in
    // case 1 above.
    let slengthmod := mod(slength, 32)
    let mlengthmod := mod(mlength, 32)
    let submod := sub(32, slengthmod)
    let mc := add(_postBytes, submod)
    let end := add(_postBytes, mlength)
    let mask := sub(exp(0x100, submod), 1)

    sstore(sc, add(sload(sc), and(mload(mc), mask)))

    for {
      sc := add(sc, 1)
      mc := add(mc, 0x20)
    } lt(mc, end) {
      sc := add(sc, 1)
```

```
            mc := add(mc, 0x20)
          } {
            sstore(sc, mload(mc))
          }

          mask := exp(0x100, sub(mc, end))

          sstore(sc, mul(div(mload(mc), mask), mask))
        }
      }
    }

    function slice(bytes memory _bytes, uint256 _start, uint256 _length) internal pure returns
  (bytes memory) {
      require(_length + 31 >= _length, "slice_overflow");
      require(_bytes.length >= _start + _length, "slice_outOfBounds");

      bytes memory tempBytes;

      assembly {
        switch iszero(_length)
        case 0 {
          // Get a location of some free memory and store it in tempBytes as
          // Solidity does for memory variables.
          tempBytes := mload(0x40)

          // The first word of the slice result is potentially a partial
          // word read from the original array. To read it, we calculate
          // the length of that partial word and start copying that many
          // bytes into the array. The first word we copy will start with
          // data we don't care about, but the last `lengthmod` bytes will
```

```
    // land at the beginning of the contents of the new array. When
    // we're done copying, we overwrite the full first word with
    // the actual length of the slice.
    let lengthmod := and(_length, 31)

    // The multiplication in the next line is necessary
    // because when slicing multiples of 32 bytes (lengthmod == 0)
    // the following copy loop was copying the origin's length
    // and then ending prematurely not copying everything it should.
    let mc := add(add(tempBytes, lengthmod), mul(0x20, iszero(lengthmod)))
    let end := add(mc, _length)

    for {
      // The multiplication in the next line has the same exact purpose
      // as the one above.
      let cc := add(add(add(_bytes, lengthmod), mul(0x20, iszero(lengthmod))), _start)
    } lt(mc, end) {
      mc := add(mc, 0x20)
      cc := add(cc, 0x20)
    } {
      mstore(mc, mload(cc))
    }

    mstore(tempBytes, _length)

    //update free-memory pointer
    //allocating the array padded to 32 bytes like the compiler does now
    mstore(0x40, and(add(mc, 31), not(31)))
  }
  //if we want a zero-length slice let's just return a zero-length array
  default {
```

```
            tempBytes := mload(0x40)

            //zero out the 32 bytes slice we are about to return

            //we need to do it because Solidity does not garbage collect

            mstore(tempBytes, 0)


            mstore(0x40, add(tempBytes, 0x20))

          }

        }


        return tempBytes;

    }


    function toAddress(bytes memory _bytes, uint256 _start) internal pure returns (address) {

        require(_bytes.length >= _start + 20, "toAddress_outOfBounds");

        address tempAddress;


        assembly {

          tempAddress := div(mload(add(add(_bytes, 0x20), _start)),
0x1000000000000000000000000)

        }


        return tempAddress;

    }


    function toUint8(bytes memory _bytes, uint256 _start) internal pure returns (uint8) {

        require(_bytes.length >= _start + 1, "toUint8_outOfBounds");

        uint8 tempUint;


        assembly {

          tempUint := mload(add(add(_bytes, 0x1), _start))

        }
```

```
    return tempUint;
}

function toUint16(bytes memory _bytes, uint256 _start) internal pure returns (uint16) {
    require(_bytes.length >= _start + 2, "toUint16_outOfBounds");
    uint16 tempUint;

    assembly {
        tempUint := mload(add(add(_bytes, 0x2), _start))
    }

    return tempUint;
}

function toUint32(bytes memory _bytes, uint256 _start) internal pure returns (uint32) {
    require(_bytes.length >= _start + 4, "toUint32_outOfBounds");
    uint32 tempUint;

    assembly {
        tempUint := mload(add(add(_bytes, 0x4), _start))
    }

    return tempUint;
}

function toUint64(bytes memory _bytes, uint256 _start) internal pure returns (uint64) {
    require(_bytes.length >= _start + 8, "toUint64_outOfBounds");
    uint64 tempUint;

    assembly {
```

```
        tempUint := mload(add(add(_bytes, 0x8), _start))
    }

    return tempUint;
}

function toUint96(bytes memory _bytes, uint256 _start) internal pure returns (uint96) {
    require(_bytes.length >= _start + 12, "toUint96_outOfBounds");
    uint96 tempUint;

    assembly {
        tempUint := mload(add(add(_bytes, 0xc), _start))
    }

    return tempUint;
}

function toUint128(bytes memory _bytes, uint256 _start) internal pure returns (uint128) {
    require(_bytes.length >= _start + 16, "toUint128_outOfBounds");
    uint128 tempUint;

    assembly {
        tempUint := mload(add(add(_bytes, 0x10), _start))
    }

    return tempUint;
}

function toUint256(bytes memory _bytes, uint256 _start) internal pure returns (uint256) {
    require(_bytes.length >= _start + 32, "toUint256_outOfBounds");
    uint256 tempUint;
```

```
        assembly {
            tempUint := mload(add(add(_bytes, 0x20), _start))
        }

        return tempUint;
    }

    function toBytes32(bytes memory _bytes, uint256 _start) internal pure returns (bytes32) {
        require(_bytes.length >= _start + 32, "toBytes32_outOfBounds");
        bytes32 tempBytes32;

        assembly {
            tempBytes32 := mload(add(add(_bytes, 0x20), _start))
        }

        return tempBytes32;
    }

    function equal(bytes memory _preBytes, bytes memory _postBytes) internal pure returns
(bool) {
        bool success = true;

        assembly {
            let length := mload(_preBytes)

            // if lengths don't match the arrays are not equal
            switch eq(length, mload(_postBytes))
            case 1 {
                // cb is a circuit breaker in the for loop since there's
                //  no said feature for inline assembly loops
```

```
        // cb = 1 - don't breaker
        // cb = 0 - break
        let cb := 1

        let mc := add(_preBytes, 0x20)
        let end := add(mc, length)

        for {
          let cc := add(_postBytes, 0x20)
          // the next line is the loop condition:
          // while(uint256(mc < end) + cb == 2)
        } eq(add(lt(mc, end), cb), 2) {
          mc := add(mc, 0x20)
          cc := add(cc, 0x20)
        } {
          // if any of these checks fails then arrays are not equal
          if iszero(eq(mload(mc), mload(cc))) {
            // unsuccess:
            success := 0
            cb := 0
          }
        }
      }
      default {
        // unsuccess:
        success := 0
      }
    }

    return success;
  }
```

```
  function equal_nonAligned(bytes memory _preBytes, bytes memory _postBytes) internal pure
returns (bool) {
    bool success = true;

    assembly {
      let length := mload(_preBytes)

      // if lengths don't match the arrays are not equal
      switch eq(length, mload(_postBytes))
      case 1 {
        // cb is a circuit breaker in the for loop since there's
        //  no said feature for inline assembly loops
        // cb = 1 - don't breaker
        // cb = 0 - break
        let cb := 1

        let endMinusWord := add(_preBytes, length)
        let mc := add(_preBytes, 0x20)
        let cc := add(_postBytes, 0x20)

        for {
          // the next line is the loop condition:
          // while(uint256(mc < endWord) + cb == 2)
        } eq(add(lt(mc, endMinusWord), cb), 2) {
          mc := add(mc, 0x20)
          cc := add(cc, 0x20)
        } {
          // if any of these checks fails then arrays are not equal
          if iszero(eq(mload(mc), mload(cc))) {
            // unsuccess:
```

```
            success := 0
            cb := 0
        }
    }

    // Only if still successful
    // For <1 word tail bytes
    if gt(success, 0) {
        // Get the remainder of length/32
        // length % 32 = AND(length, 32 - 1)
        let numTailBytes := and(length, 0x1f)
        let mcRem := mload(mc)
        let ccRem := mload(cc)
        for {
            let i := 0
            // the next line is the loop condition:
            // while(uint256(i < numTailBytes) + cb == 2)
        } eq(add(lt(i, numTailBytes), cb), 2) {
            i := add(i, 1)
        } {
            if iszero(eq(byte(i, mcRem), byte(i, ccRem))) {
                // unsuccess:
                success := 0
                cb := 0
            }
        }
    }
}
default {
    // unsuccess:
    success := 0
```

```
        }
      }

      return success;
    }

    function equalStorage(bytes storage _preBytes, bytes memory _postBytes) internal view
returns (bool) {
      bool success = true;

      assembly {
        // we know _preBytes_offset is 0
        let fslot := sload(_preBytes.slot)
        // Decode the length of the stored array like in concatStorage().
        let slength := div(and(fslot, sub(mul(0x100, iszero(and(fslot, 1))), 1)), 2)
        let mlength := mload(_postBytes)

        // if lengths don't match the arrays are not equal
        switch eq(slength, mlength)
        case 1 {
          // slength can contain both the length and contents of the array
          // if length < 32 bytes so let's prepare for that
          // v. http://solidity.readthedocs.io/en/latest/miscellaneous.html#layout-of-state-
variables-in-storage
          if iszero(iszero(slength)) {
            switch lt(slength, 32)
            case 1 {
              // blank the last byte which is the length
              fslot := mul(div(fslot, 0x100), 0x100)

              if iszero(eq(fslot, mload(add(_postBytes, 0x20)))) {
                // unsuccess:
```

```
        success := 0
    }
}
default {
    // cb is a circuit breaker in the for loop since there's
    //  no said feature for inline assembly loops
    // cb = 1 - don't breaker
    // cb = 0 - break
    let cb := 1

    // get the keccak hash to get the contents of the array
    mstore(0x0, _preBytes.slot)
    let sc := keccak256(0x0, 0x20)

    let mc := add(_postBytes, 0x20)
    let end := add(mc, mlength)

    // the next line is the loop condition:
    // while(uint256(mc < end) + cb == 2)
    for {

    } eq(add(lt(mc, end), cb), 2) {
        sc := add(sc, 1)
        mc := add(mc, 0x20)
    } {
        if iszero(eq(sload(sc), mload(mc))) {
            // unsuccess:
            success := 0
            cb := 0
        }
    }
```

```
            }
          }
        }
        default {
          // unsuccess:
          success := 0
        }
      }

      return success;
    }
}
```

/// @title Encrypted Address Library

/// @notice Provides methods for creating and managing addresses encrypted with FHE (Fully Homomorphic Encryption)

/// @dev Assumes the existence of an FHE library that implements fully homomorphic encryption functions

/// @dev A representation of an encrypted address using Fully Homomorphic Encryption.

/// It consists of 5 encrypted 32-bit unsigned integers (`euint32`).

```
struct Eaddress {
  euint32[5] values;
}
```

library ConfAddress {

  /// @notice Encrypts a plaintext Ethereum address into its encrypted representation (`eaddress`).

  /// @dev Iterates over 5 chunks of the address, applying a bitmask to each, then encrypting with `FHE`.

  /// @param addr The plain Ethereum address to encrypt

  /// @return eaddr The encrypted representation of the address

```
  function toEaddress(address addr) internal pure returns (Eaddress memory) {
    uint160 addrValue = uint160(address(addr));
    /// @dev A bitmask constant for selecting specific 32-bit chunks from a 160-bit Ethereum address.
    /// It has the first 32 bits set to 1, and the remaining bits set to 0.
    uint160 MASK = uint160(uint256(0x000000000000000000000000FFFFFFFF00000000000000000000000000000000));

    Eaddress memory eaddr;

    for (uint i = 0; i < 5; i++) {
      uint160 currentChunkOffset = uint160(i * 32);
      uint160 mask = MASK >> currentChunkOffset; // Mask the correct chunk based on i
      uint32 chunk = uint32((addrValue & mask) >> (128 - currentChunkOffset));
      eaddr.values[i] = FHE.asEuint32(chunk);
    }

    return eaddr;
  }

  /// @notice Decrypts an `eaddress` to retrieve the original plaintext Ethereum address.
  /// @dev This operation should be used with caution as it exposes the encrypted address.
  /// @param eaddr The encrypted address to decrypt
  /// @return The decrypted plaintext Ethereum address
  function unsafeToAddress(Eaddress memory eaddr) internal pure returns (address) {
    uint160 addrValue;
    for (uint i = 0; i < 5; i++) {
      uint32 currentChunkOffset = uint32((4 - i) * 32);
      uint32 val = FHE.decrypt(eaddr.values[i]);
      uint160 currentValue = uint160(val) << currentChunkOffset;
      addrValue += currentValue;
```

```
    }

    bytes memory addrBz = new bytes(32);
    assembly {
      mstore(add(addrBz, 32), addrValue)
    }

    return BytesLib.toAddress(addrBz, 12);
  }

  /// @notice Re-encrypts the encrypted values within an ` eaddress` .
  /// @dev The re-encryption is done to change the encrypted representation without
  /// altering the underlying plaintext address, which can be useful for obfuscation purposes in storage.
  /// @param eaddr The encrypted address to re-encrypt
  /// @param ezero An encrypted zero value that triggers the re-encryption
  function resestEaddress(Eaddress memory eaddr, euint32 ezero) internal pure {
    for (uint i = 0; i < 5; i++) {
      // Adding zero will practiaclly reencrypt the value without it being changed
      eaddr.values[i] = eaddr.values[i] + ezero;
    }
  }

  /// @notice Determines if an encrypted address is equal to a given plaintext Ethereum address.
  /// @dev This operation encrypts the plaintext address and compares the encrypted representations.
  /// @param lhs The encrypted address to compare
  /// @param addr The plaintext Ethereum address to compare against
  /// @return res A boolean indicating if the encrypted and plaintext addresses are equal
  function equals(Eaddress storage lhs, address payable addr) internal view returns (ebool) {
    Eaddress memory rhs = toEaddress(addr);
```

```
    ebool res = FHE.eq(lhs.values[0], rhs.values[0]);
    for (uint i = 1; i < 5; i++) {
      res = res & FHE.eq(lhs.values[i], rhs.values[i]);
    }

    return res;
  }

  function conditionalUpdate(
    ebool condition,
    Eaddress memory eaddr,
    Eaddress memory newEaddr
  ) internal pure returns (Eaddress memory) {
    for (uint i = 0; i < 5; i++) {
      // Even if condition is false the ENCRYPTED value of eaddr.values[i] will be changed
      // because the encryption is not deterministic
      // so no one will know whether the highest bidder was changed or not
      eaddr.values[i] = FHE.select(condition, newEaddr.values[i], eaddr.values[i]);
    }

    return eaddr;
  }
}

struct HistoryEntry {
  euint32 amount;
  bool refunded;
}

contract Auction is Permissioned {
  address payable public auctioneer;
```

```
mapping(address => HistoryEntry) internal auctionHistory;
euint32 internal CONST_0_ENCRYPTED;
euint32 internal highestBid;
Eaddress internal defaultAddress;
Eaddress internal highestBidder;
euint32 internal eMaxEuint32;
uint256 public auctionEndTime;
IFHERC20 internal _wfhenix;
address internal NO_BID_ADDRESS;

// When auction is ended this will contain the PLAINTEXT winner address
address public winnerAddress;

event AuctionEnded(address winner, uint32 bid);

constructor(address wfhenix, uint256 biddingTime) payable {
  _wfhenix = IFHERC20(wfhenix);
  auctioneer = payable(msg.sender);
  auctionEndTime = block.timestamp + biddingTime;
  CONST_0_ENCRYPTED = FHE.asEuint32(0);
  highestBid = CONST_0_ENCRYPTED;
  NO_BID_ADDRESS = 0xFFfFfFffFFfffFFfFFfFFFFFffFFFffffFfFFFfF;
  for (uint i = 0; i < 5; i++) {
    defaultAddress.values[i] = CONST_0_ENCRYPTED;
    highestBidder.values[i] = CONST_0_ENCRYPTED;
  }

  eMaxEuint32 = FHE.asEuint32(0xFFFFFFFF);
}

// Modifiers
```

```
modifier onlyAuctioneer() {
   require(msg.sender == auctioneer, "Only auctioneer can perform this action");
   _;
}

modifier afterAuctionEnds() {
   require(block.timestamp >= auctionEndTime, "Auction ongoing");
   _;
}

modifier auctionNotEnded() {
   require(winnerAddress == address(0), "Auction not ended");
   _;
}

modifier auctionEnded() {
   require(winnerAddress != address(0), "Auction already ended");
   _;
}

modifier notWinner() {
   require(msg.sender != winnerAddress, "Winner cannot perform this action");
   _;
}

function updateHistory(address addr, euint32 currentBid) internal returns (euint32) {
   // Check for overflow, if such, just don't change the actualBid
   // NOTE: overflow is most likely an abnormal action so the funds WON'T be refunded!
   if (!FHE.isInitialized(auctionHistory[addr].amount)) {
      HistoryEntry memory entry;
      entry.amount = currentBid;
```

```
        entry.refunded = false;
        auctionHistory[addr] = entry;
        return auctionHistory[addr].amount;
      }

      // Checking overflow here is optional as in real-life precision would be accounted for.
      ebool hadOverflow = (eMaxEuint32 - currentBid).lt(auctionHistory[addr].amount);
      euint32 actualBid = FHE.select(hadOverflow, CONST_0_ENCRYPTED, currentBid);

      // Add the actual bid to the previous bid
      // If there was no bid it will work because the default value of uint32 is encrypted 02
      auctionHistory[addr].amount = auctionHistory[addr].amount + actualBid;
      return auctionHistory[addr].amount;
    }

    function bid(inEuint32 calldata amount)
    external
    auctionNotEnded
    {

      euint32 spent = _wfhenix.transferFromEncrypted(msg.sender, address(this), amount);

      euint32 newBid = updateHistory(msg.sender, spent);
      // Can't update here highestBid directly because we need and indication whether the
highestBid was changed
      // if we will change here the highestBid
      // we will have an edge case when the current bid will be equal to the highestBid
      euint32 newHeighestBid = FHE.max(newBid, highestBid);

      Eaddress memory eaddr = ConfAddress.toEaddress(payable(msg.sender));
      ebool wasBidChanged = newHeighestBid.gt(highestBid);
```

```
    highestBidder = ConfAddress.conditionalUpdate(wasBidChanged, highestBidder, eaddr);
    highestBid = newHeighestBid;
}

function getMyBidDebug (address account)
external
view
returns (uint256) {
    return FHE.decrypt(auctionHistory[account].amount);
}

function getMyBid (address account, Permission memory auth)
external
view
onlyPermitted(auth, account)
returns (uint256) {
    return FHE.decrypt(auctionHistory[account].amount);
}

function getWinner()
external
view
auctionEnded
returns (address) {
    return winnerAddress;
}

function getWinningBid()
external
view
```

```
auctionEnded
returns (uint256, address) {
  return (FHE.decrypt(highestBid), winnerAddress);
}

function endAuction()
external
onlyAuctioneer
afterAuctionEnds
auctionNotEnded
{
  winnerAddress = ConfAddress.unsafeToAddress(highestBidder);
  if (winnerAddress == address(0)) {
   winnerAddress = NO_BID_ADDRESS;
  }
  // The cards can be revealed now, we can safely reveal the bidder
  emit AuctionEnded(winnerAddress, FHE.decrypt(highestBid));
}

// just for debugging purposes
function debugEndAuction()
public
onlyAuctioneer
auctionNotEnded
{
  winnerAddress = ConfAddress.unsafeToAddress(highestBidder);
  if (winnerAddress == address(0)) {
   winnerAddress = NO_BID_ADDRESS;
  }
  // The cards can be revealed now, we can safely reveal the bidder
  emit AuctionEnded(winnerAddress, FHE.decrypt(highestBid));
```

```
  }

  function redeemFunds()
  external
  notWinner
  auctionEnded
  {
    require(!auctionHistory[msg.sender].refunded, "Already refunded");

    euint32 toBeRedeemed = auctionHistory[msg.sender].amount;

    auctionHistory[msg.sender].refunded = true;

    _wfhenix.transferEncrypted(msg.sender, toBeRedeemed);
  }
}
```

Lottery.sol

```
// SPDX-License-Identifier: BSD-3-Clause-Clear

pragma solidity >=0.8.13 <0.9.0;

import "@fhenixprotocol/contracts/FHE.sol";

contract Lottery {
  euint8 private winningNumber;
  euint32 private currentPrize;
  mapping (address => euint32) rewards;

  uint256 public endingTime;
  uint32 public ticketPrice;
```

```
    uint32 public ticketCount;

    event LotteryTicketBought(uint ticketNo);

    error TooEarly(uint timeEnding, uint timeNow);
    error TooLate(uint timeEnding, uint timeNow);
    error InsufficientPayment(uint32 price);

    constructor(uint32 _ticketPrice, inEuint8 memory initialRandom) {
        // at first, the contract deployer knows the winning number, but he can only win the prize he
put in
        // once another player buys a ticket, he doesn't know the winning number anymore
        winningNumber = FHE.asEuint8(initialRandom);

        // endingTime = block.timestamp + 10 days;
        // for testing purposes:
        endingTime = block.timestamp + 20 seconds;
        ticketPrice = _ticketPrice;
        currentPrize = FHE.asEuint32(0);
    }

    function fundPrize() public payable {
        currentPrize = FHE.add(currentPrize, FHE.asEuint32(msg.value));
    }

    function buyTicket(inEuint8 calldata encryptedGuess) public payable onlyBeforeEnd {
        if (msg.value < ticketPrice) {
            revert InsufficientPayment(ticketPrice);
        }

        euint8 guess = FHE.asEuint8(encryptedGuess);
```

```
    // add message value to prize:
    currentPrize = currentPrize.add(FHE.asEuint32(msg.value));
    ticketCount += 1;

    // check winner:
    ebool isWinner = winningNumber.eq(guess);

    // alter the next winning number - This ensures that every subsequent winningNumber will
be
    // unpredictable by someone who isn't involved with the paying party
    winningNumber = winningNumber.xor(guess);

    // store player's reward:
    rewards[msg.sender] = FHE.select(isWinner, rewards[msg.sender].add(currentPrize),
rewards[msg.sender]);
    currentPrize = FHE.select(isWinner, FHE.asEuint32(0), currentPrize);

    emit LotteryTicketBought(ticketCount);
  }

  function checkRewards(bytes32 publicKey) public view onlyAfterEnd returns (bytes memory){
    // check if I have rewards
    return FHE.sealoutput(rewards[msg.sender], publicKey);
  }

  function redeemRewards() public onlyAfterEnd {
    euint32 reward = rewards[msg.sender];
    rewards[msg.sender] = FHE.asEuint32(0);

    payable(msg.sender).transfer(FHE.decrypt(reward));
  }
```

```
    modifier onlyBeforeEnd() {
        if (block.timestamp >= endingTime) revert TooLate(endingTime, block.timestamp);
        _;
    }

    modifier onlyAfterEnd() {
        if (block.timestamp < endingTime) revert TooEarly(endingTime, block.timestamp);
        _;
    }
}
```

Voting.sol

```
// SPDX-License-Identifier: BSD-3-Clause-Clear

pragma solidity >=0.8.19 <0.9.0;

// import "@fhenixprotocol/contracts/FHE.sol";
import "./FHE.sol";
import "@fhenixprotocol/contracts/access/Permission.sol";

contract Voting is Permissioned {
    uint8 internal constant MAX_OPTIONS = 4;

    // Pre-compute these to prevent unnecessary gas usage for the users
    // euint16 internal _zero = FHE.asEuint16(0);
    // euint16 internal _one = FHE.asEuint16(1);
    euint32 internal _u32Sixteen = FHE.asEuint32(16);
    euint8[MAX_OPTIONS] internal _encOptions = [FHE.asEuint8(0), FHE.asEuint8(1),
FHE.asEuint8(2), FHE.asEuint8(3)];
```

```
  string public proposal;

  string[] public options;

  uint public voteEndTime;

  euint16[MAX_OPTIONS] internal _tally; // Since every vote is worth 1, I assume we can use a
16-bit integer


  euint8 internal _winningOption;

  euint16 internal _winningTally;


  mapping(address => euint8) internal _votes;


  constructor(string memory _proposal, string[] memory _options, uint votingPeriod) {

    require(options.length <= MAX_OPTIONS, "too many options!");


    proposal = _proposal;

    options = _options;

    voteEndTime = block.timestamp + votingPeriod;

  }


  function vote(inEuint8 memory voteBytes) public {

    require(block.timestamp < voteEndTime, "voting is over!");

    require(!FHE.isInitialized(_votes[msg.sender]), "already voted!");

    euint8 encryptedVote = FHE.asEuint8(voteBytes); // Cast bytes into an encrypted type

    _requireValid(encryptedVote);


    _votes[msg.sender] = encryptedVote;

    _addToTally(encryptedVote /* , _one */);

  }


  function finalize() public {

    require(voteEndTime < block.timestamp, "voting is still in progress!");
```

```
    _winningOption = _encOptions[0];

    _winningTally = _tally[0];

    for (uint8 i = 1; i < options.length; i++) {

      euint16 newWinningTally = FHE.max(_winningTally, _tally[i]);

      _winningOption = FHE.select(newWinningTally.gt(_winningTally), _encOptions[i],
_winningOption);

      _winningTally = newWinningTally;

    }

  }


  function winning() public view returns (uint8, uint16) {

    require(voteEndTime < block.timestamp, "voting is still in progress!");

    return (FHE.decrypt(_winningOption), FHE.decrypt(_winningTally));

  }


  function getUserVote(

    Permission memory signature

  ) public view onlySignedPublicKey(signature) returns (bytes memory) {

    require(FHE.isInitialized(_votes[msg.sender]), "no vote found!");

    return FHE.sealoutput(_votes[msg.sender], signature.publicKey);

  }


  function _requireValid(euint8 encryptedVote) internal view {

    // Make sure that: (0 <= vote <= options.length)

    ebool isValid = encryptedVote.gte(_encOptions[0]) &
encryptedVote.lte(_encOptions[options.length - 1]);

    FHE.req(isValid);

  }


  function _addToTally(euint8 option /* , euint16 amount */) internal {

    // We don't want to leak the user's vote, so we have to change the tally of every option.
```

```
        // So for example, if the user voted for option 1:
        // tally[0] = tally[0] + enc(0)
        // tally[1] = tally[1] + enc(1)
        // etc ..
        for (uint8 i = 0; i < options.length; i++) {
          // euint16 amountOrZero = FHE.select(option.eq(_encOptions[i]), _one, _zero);
          ebool amountOrZero = option.eq(_encOptions[i]); // `eq()` result is known to be enc(0) or enc(1)
          _tally[i] = _tally[i] + amountOrZero.toU16(); // `eq()` result is known to be enc(0) or enc(1)
        }
    }
}
```

WrappingERC20.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import { FHERC20 } from "@fhenixprotocol/contracts/experimental/token/FHERC20/FHERC20.sol";
import { FHE, euint32, inEuint32 } from "@fhenixprotocol/contracts/FHE.sol";

contract ExampleToken is FHERC20 {
    constructor(string memory name, string memory symbol)
      FHERC20(
        bytes(name).length == 0 ? "FHE Token" : name,
        bytes(symbol).length == 0 ? "FHE" : symbol
      ) {}

      function mint(uint256 amount) public {
        _mint(msg.sender, amount);
      }
```

```
    function mintEncrypted(inEuint32 calldata encryptedAmount) public {
        euint32 amount = FHE.asEuint32(encryptedAmount);
        if (!FHE.isInitialized(_encBalances[msg.sender])) {
            _encBalances[msg.sender] = amount;
        } else {
            _encBalances[msg.sender] = _encBalances[msg.sender] + amount;
        }

        totalEncryptedSupply = totalEncryptedSupply + amount;
    }
}
```